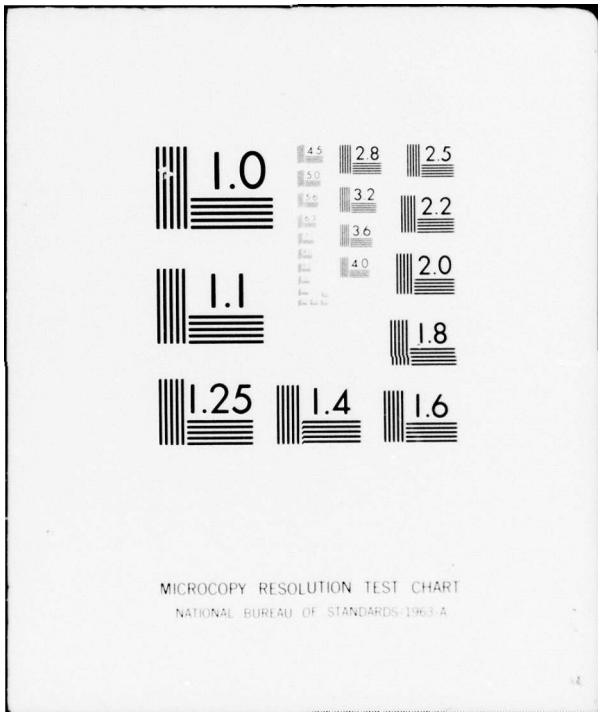
AD-A041 048 GENERAL RESEARCH CORP SANTA BARBARA CALIF F/G 9/2
JAVS TECHNICAL REPORT. METHODOLOGY REPORT.(U)
APR 77 N B BROOKS, C GANNON F30602-76-C-0233
UNCLASSIFIED RADC-TR-77-126-VOL-3 NL

1 OF 1
AD A041048

END
DATE FILMED
7-77


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

(12) NW

RADC-TR-77-126, Volume III (of three)
Final Technical Report
April 1977

JAVS TECHNICAL REPORT
Methodology Report

General Research Corporation

Approved for public release; distribution unlimited.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report contains a large percentage of machine-produced copy which is not of the highest printing quality but because of economical consideration, it was determined in the best interest of the government that they be used in this publication.

This report has been reviewed by the RADC Information Office (OI) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

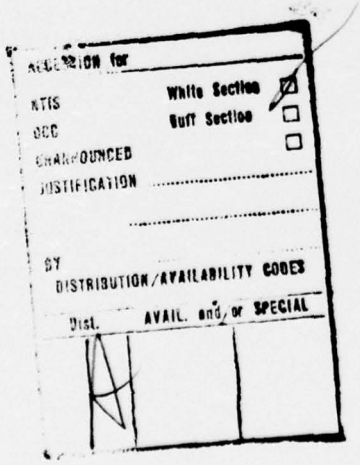
This report has been reviewed and is approved for publications.

APPROVED: Frank S LaMonica

FRANK S. LAMONICA
Project Engineer

APPROVED: Robert D. Krutz

ROBERT D. KRUTZ, Colonel, USAF
Chief, Information Sciences Division



FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: RADC-TR-77-126, Volume III (of three)	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE *(and Subtitle)*: JAVS TECHNICAL REPORT Methodology Report	5. TYPE OF REPORT & PERIOD COVERED: Final Technical Report May 76 - Dec 76
	6. PERFORMING ORG. REPORT NUMBER: N/A
7. AUTHOR(s): N. B. Brooks C. Gannon	8. CONTRACT OR GRANT NUMBER(s): F30602-76-C-0233
9. PERFORMING ORGANIZATION NAME AND ADDRESS: General Research Corporation P. O. Box 3587 Santa Barbara CA 93105	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 63728F 55500838
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (ISIM) Griffiss AFB NY 13441	12. REPORT DATE: April 1977 13. NUMBER OF PAGES: 87
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*: Same	15. SECURITY CLASS. *(of this report)*: UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer:
Frank LaMonica (ISIM)

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Computer software
Software testing
Software verification
JAVS
Automated Verification System

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

The JOVIAL Automated Verification System (JAVS) is a tool for analyzing source programs written in the J3 dialect of the JOVIAL language. From the user's viewpoint, JAVS consists of a sequence of processing steps which (1) analyze his JOVIAL source text, (2) guide him in preparing test cases for his programs, (3) analyze the results of tests executed by his programs, and (4) automatically document his programs.

(cont'd)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

This report describes the application of a testing methodology utilizing an Automated Verification System (AVS) such as JAVS. Sections of this report present an overview of the testing methodology and the capabilities of JAVS and describe actual testing experience with JAVS, the general role of an AVS in applying the testing methodology, practical techniques for particular test situations, and expanded capabilities for advanced AVS implementations.

LIST OF JAVS REPORTS

- JAVS Technical Report: Vol. 1, User's Guide. This report is an introduction to using JAVS in the testing process. Its primary purpose is to acquaint the user with the innate potential of JAVS to aid in the program testing process so that an efficient approach to program verification can be undertaken. Only the basic principles by which JAVS provides this assistance are discussed. These give the user a level of understanding necessary to see the utility of the system. The material on JAVS processing in the report is presented in the order normally followed by the beginning JAVS user. Adequate testing can be achieved using JAVS macro commands and the job streams presented in this guide. The Appendices include a summary of all JAVS commands and a description of JAVS operation at RADC with both sample command sets and sample job control statements.

- JAVS Technical Report: Vol. 2, Reference Manual. This report describes in detail JAVS processing and each of the JAVS commands. The Reference Manual is intended to be used along with the User's Guide which contains the machine-dependent information such as job control cards and file allocation. Throughout the Reference Manual, modules from a sample JOVIAL program are used in the examples. Each JAVS command is explained in detail, and a sample of each report produced by JAVS is included with the appropriate command. The report is organized into two major parts: one describing the JAVS system and the other containing the description of each JAVS command in alphabetical order. The Appendices include a complete listing of all error messages directly produced by JAVS processing.

- JAVS Technical Report: Vol. 3, Methodology Report. This report describes the methodology which underlies and is supported by JAVS. The methodology is tailored to be largely independent of implementation and language. The discussion in the text is intended to be intuitive and demonstrative. Some of the methodology is based upon the experience of using JAVS to test a large information management system. A long-term growth path for automated verification systems that supports the methodology is described.

- JAVS Computer Program Documentation: Vol. 1, System Design and Implementation. This report contains a description of JAVS software design, the organization and contents of the JAVS data base, and a description of the software for each JAVS component: its function, each of the modules in the component, and the global data structures used by the component. The report is intended primarily as an informal reference for use in JAVS software maintenance as a companion to the Software Analysis reports described below. Included in the appendices are the templates for probe code inserted by instrumentation processing for both structural and directive instrumentation and an alphabetical list of all modules in the system (including system routines) with the formal parameters and data type of each parameter.

- JAVS Computer Program Documentation: Vol. 2, Software Analysis. This volume is a collection of computer output produced by JAVS standard processing steps. The source for each component of the JAVS software has been analyzed

to produce enhanced source listings of JAVS with indentation and control structure identification, inter-module dependence, all module invocations with formal and actual parameters, module control structure, a cross reference of symbol usage, tree report for each leading module, and report showing size of each component. It is intended to be used with the System Design and Implementation Manual for JAVS software maintenance. The Software Analysis reports, on file at RADC, are an excellent example of the use of JAVS for computer software documentation.

- JAVS Preprocessor for JOVIAL. This report, prepared for GRC by its subcontractor, System Development Corporation (SDC), describes the software for the JAVS-2 component: its origin as the GEN1 part of the SAM-D ED Compiler, the modifications made in GEN1 to adapt the code for JAVS-2, the JAVS-2 code modules, and the data structures. It contains excerpts of other SDC reports on the SAM-D ED JOVIAL Compiler System. The report reflects the status of the software for JAVS-2 as delivered by SDC to GRC in September 1974. The description of JAVS-2 software contained in the System Design and Integration report reflects the status of JAVS-2 as delivered to RADC by GRC in September 1975 and thereby supercedes the SDC report.

- JAVS Final Report. The final report for the project describes the implementation and application of a methodology for systematically and comprehensively testing computing software. The methodology utilizes the structure of the software undergoing test as the basis for anlaysis by an automated verification system (AVS). The report also evaluates JAVS as a tool for software development and testing.

CONTENTS

ILLUSTRATIONS

EVALUATION

The purpose of this effort was to enhance the JOVIAL Automated Verification System (JAVS) and to implement a systematic software testing program using the JAVS to assist in the testing process. Developed to aid in the testing and verification of JOVIAL J3 programs, it provides the ability to increase the practical reliability of software by increasing the achieved level of testing. As a result of this effort, the JAVS was enhanced and successfully tuned for operational use. This report is the third of a series of three volumes which provide excellent supporting documentation on its application and use.

Frank S. LaMonica

FRANK S. LAMONICA
Project Engineer

1 SUMMARY

At present, computer software is tested only according to its developers' intuitions, if it is tested at all. The reliability of software is at least partially dependent on the thoroughness of its testing; increased testing therefore contributes to increased reliability.

Simple computer programs can be comprehensively tested without difficulty by inspection. However, when computer software becomes so complex that human intuition is inadequate to deal with its subtleties, the testing must be based on a systematic and rigorous methodology. One purpose of such a methodology is to keep the cost of testing--in money and in time--to a tolerable level.

Under a previous contract,[1] GRC developed and implemented a methodology for testing computer software.[2] It is designed to be largely independent of implementation and programming language. The methodology incorporates an Automated Verification System (AVS) which analyzes the structure of the software, in its source-text form.

The AVS consists of an integrated series of tools designed to:

- Measure the effectiveness of software test cases, both individually and cumulatively
- Facilitate the construction of test data that will thoroughly exercise the software
- Analyze the requirements for retesting after modification of the software

In that contract, an implementation of the methodology was developed for programs written in the J3 dialect of the JOVIAL language.[3] The resulting AVS, JAVS (for JOVIAL Automated Verification System), is operational on both the Honeywell Information Systems HIS 6180 at Rome Air Development Center and the Control Data Corporation CDC 6400 at General Research Corporation.[4,5]

This report describes the application of the testing methodology for systematically and comprehensively testing computer software. The techniques described in the following sections were employed in the testing conducted during the previous contract and its present follow-on. The present contract[6] has as its objectives:

- Evaluate the concept of automated tools to support testing in general, and the present capabilities of JAVS in particular
- Accumulate experience in using automated tools
- Refine the methodology for testing with AVS assistance, and develop design goals for extended AVS capability to support the methodology
- Increase the confidence in operational programs through systematic testing using automated tools, and gather data on the actual (and potential) contribution of an AVS to the testing process
- Document the results of the tests, and the refined methodology and projected AVS design

As part of the JAVS acceptance tests[7] the first contract used as software test objects (1) a small, complete engineering application program called BLSTIC and (2) the software for JAVS itself, a large program with well-defined structure. The test object for the current contract is the COMPOSE segment of SAC's Force Management Information System (FMIS). With each of these test objects, we have experienced the increased power that an AVS gives the tester in achieving software test objectives. An AVS offers a totally new dimension in software engineering, not only in software testing but also as an invaluable assistant in development and maintenance.

An important outcome of the test activities thus far is the enormous value of high-quality, thorough, up-to-date, and accurate documentation of the test objects which is automatically produced by the AVS as part of applying the methodology.

Although this report is largely self-contained, the reader is encouraged to make use of Refs. 4 and 5 for their extensive examples of AVS reports and of Ref. 2 for the conceptual basis and detailed description of the methodology. The emphasis here is upon the application of the testing methodology with JAVS (Secs. 4 and 5) and forecasting the capabilities needed in more powerful automated test tools (Sec. 6). The sections in sequential order describe:

- An overview of the testing methodology (Sec. 2)
- An overview of JAVS capabilities (Sec. 3)
- An overview of the testing experience (Sec. 4)
- Application of the testing methodology (Sec. 5)
- Advanced AVS capabilities (Sec. 6)

Section 2 is largely taken from the Final Report of the first contract.[8] It is included here to make this report self-contained. Section 3 presents a high-level overview of the capabilities of JAVS; detailed descriptions of these capabilities are contained in Refs. 4 and 5. Section 4 reviews the testing experience with JAVS in terms of the techniques used in applying the methodology to three specific programs. The remaining sections generalize the application of the methodology and outline expanded capabilities for advanced AVS implementations.

Other reports detailing the methodology, the capabilities of JAVS, and testing experience are included in Refs. 2, 4, 5, 7, 8, and 9. Other implementations of the testing methodology are now operational for FORTRAN,[10] JOVIAL J3B,[11] and Pascal.[12]

2 OVERVIEW OF THE METHODOLOGY

This section presents an overview of the methodology as developed under the previous contract.

2.1 ISSUES IN PROGRAM TESTING

In many applications which involve a digital computer, the difficult problem of developing the software portions of the system has been of increasing concern to system managers. This is in sharp contrast to computer hardware reliability problems which can be attacked with conventional engineering techniques. For all practical purposes, computer hardware can be made as reliable as necessary through multiple redundancy and other techniques. The situation is very different for computer software, particularly in critical applications. To date there have been no truly effective ways to make software a low-risk element of a system implementation, regardless of the effort applied.

2.1.1 Approaches to Software Quality

The computer science community recognizes the software reliability problem and is developing systematic approaches to increase software reliability and, if possible, simultaneously reduce its overall cost.

Among the current "synthesis" approaches are the following:

- Structured Programming disciplines, which seek to minimize the complexity of software (and thereby enhance its overall quality and reliability) by constraining the control structures of the programming language used.
- Chief Programmer Teams, a management technique which assigns a talented person (the Chief Programmer) sole responsibility for all aspects of a software system, including its ultimate effectiveness and reliability.
- Technologically sophisticated software design methodologies such as "top down" and "bottom up" design and implementation disciplines, which attempt to systematize the software production process and thereby enhance program quality.

These "synthesis" techniques generally try to increase software quality by keeping software problems from happening in the first place.

The alternative of dealing with software which has already been developed (or is in the final stages of development) involves two primary "analysis" approaches:

- Program proofs, which demonstrate the correctness of programs by treating them as if they were mathematical theorems. A mechanical theorem prover is often used to assist in the proof construction.
- Automated Verification Systems (AVS's), which attempt to increase the practical reliability of software by raising the execution coverage of program paths.

2.1.2 Automatable Methodology

The methodology implemented in JAVS addresses only the software testing area. The need to assure a sound theoretical basis for a systematic general testing methodology is clear. It is equally important to develop a methodology that can be automated, because the combinatorial difficulties encountered in testing large-scale software systems can be so great that any purely manual systematic testing methodology would be of questionable value. Only methodologies which can be supported by an AVS are considered here; the analytical mechanisms employed are those which (by design) meet the dual requirements of generality and automatability.

2.1.3 The Meaning of "Verification"

The diagram in Fig. 2.1 shows the relationships between a software System Functional Specification, the software, and the process by which an AVS seeks to invert, or "validate," the software implementation phase, as shown by a dashed line. In Phase I, the embodied software is analyzed by the AVS to produce a set of "structurally indicated" test case patterns. In Phase II, structural indications of appropriate test cases are used to select actual test case data. The aggregate of test case data can be used to exercise the software system--that is, to assure that all portions of the software system have been exercised against some acceptance criteria.

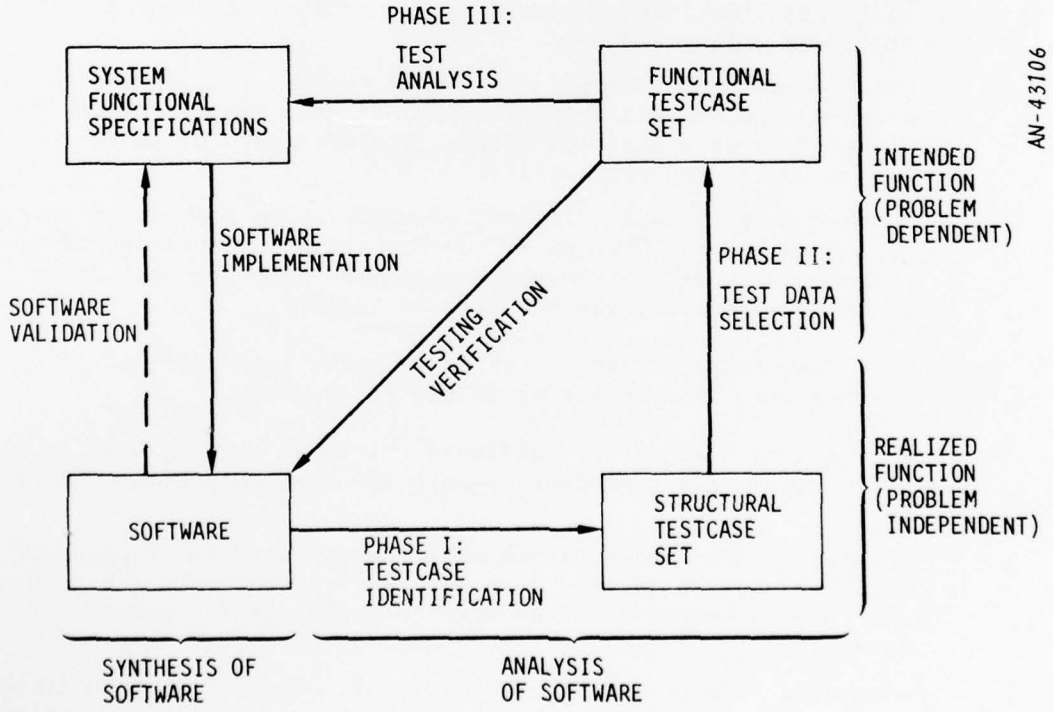


Figure 2.1. Relation Between Testing and Validation

The test case set can also be used in another important way. In Phase III the relationship between the set of functional tests and the Functional Specification is used as an indicator of the veracity of the original implementation activity. If the functional test case set does not match one-for-one with the requirements stated in the Functional Specification, then one must conclude that either the implementation is imperfect, or the specification is imperfect (assuming, of course, that the functional test case set has been correctly generated). The absence of a mismatch leads to a general increase in the program testers' belief in the software as a genuine implementation of the requirements stated in the functional specification.

2.1.4 Limitations of Testing

Although program testing is a powerful tool in this restricted sense, the current state-of-the-art will not support a fully automatable analysis of the correspondence between sets of test cases and the functional specifications of software (Phase III of the process just defined). Instead, the role of the AVS is to assure that the testing verification meets some criterion of comprehensiveness. Testing verification is the primary outcome of operation of the AVS and is an indirect indicator of the degree to which the real objective (matching tests with functional behavior) is actually met.

Comprehensive exercise of a software system does not guarantee that it is error-free. However, practical experience indicates that thorough exercise will locate a very high proportion of errors. Hence, the use of testing verification as an approximation to full program verification seems to be reasonable and practical.

2.2 SINGLE-MODULE TESTING

Single-module testing is oriented toward a particular, well-defined testing goal, which is based on the internal properties of a program's control structure: to assure that each statement in the program has been exercised at least once, and that each decision in the program has been exercised at least once to each possible outcome (but not necessarily in every possible combination).

The smallest executable piece of a program is the sequence of activities the program performs in using the outcome of a decision to determine the program's future course of action. We call this a "decision-to-decision path," or DD-path for short. A DD-path is diagrammed in Fig. 2.2.

If every DD-path in a program has been exercised at least once by a test case set for that program, then both of the testing criteria stated above have been met.

2.2.1 The Iteration Structure

During testing (after some initial set of tests), the situation is as follows. Some as-yet-untested DD-path is selected as the subject of the question "What means can be used to construct new test case data which will cause this DD-path to be executed?"

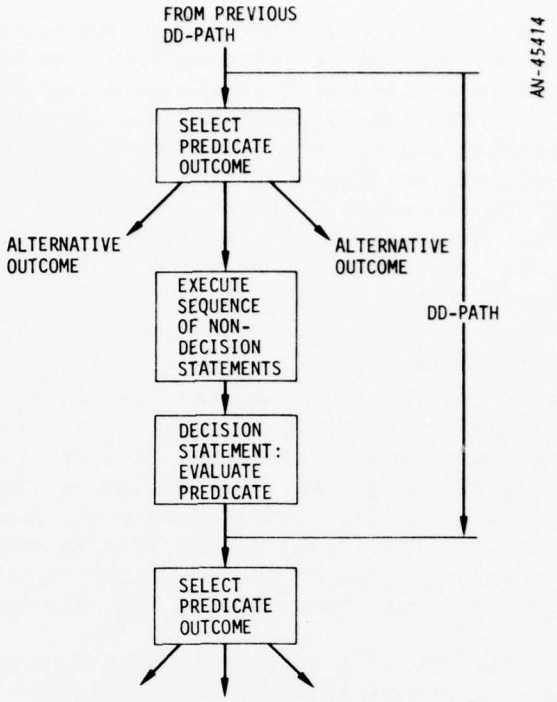


Figure 2.2. Diagram of Decision-to-Decision Path (DD-Path)

The answer to this question is provided in part by the iteration structure of a module. Briefly, the iteration structure of the module is a tree of interdependent sets of DD-paths arranged in such a way that it is easy and straightforward to identify potential program flows. The pattern needed to deal with any particular program flow is as shown in Fig. 2.3. The diagram shows a single non-iterative flow pattern modified by a single iterative flow pattern. The non-iterative flow consists of DD-paths with the labels A_1,A_2, A_3,A_4: the iteration (in this case, a simple cycle) consists of DD-paths B_1, B_2,B_3,B_4.

This pattern is a prototype of all possible patterns of program flow because it incorporates the two necessary "types" of program execution: (1) selection of future program action and (2) repetition (iteration) of previously executed actions. For any computer program, and for a particular fixed set of program "input data," the actual flow pattern which results can be represented as a collection of such patterns. Although the actual pattern for a large program may be very complex, it is always composed only of selection and iteration operations.

Single-module testing is the process of identifying a particular pattern of program flow, and then constructing input data for the module which makes that program flow pattern actually happen. A test consists of a single invocation of a module, operating in a data environment which is sufficient to satisfy the data-input needs for that invocation. The set of data needed for a test is called the test case dataset.

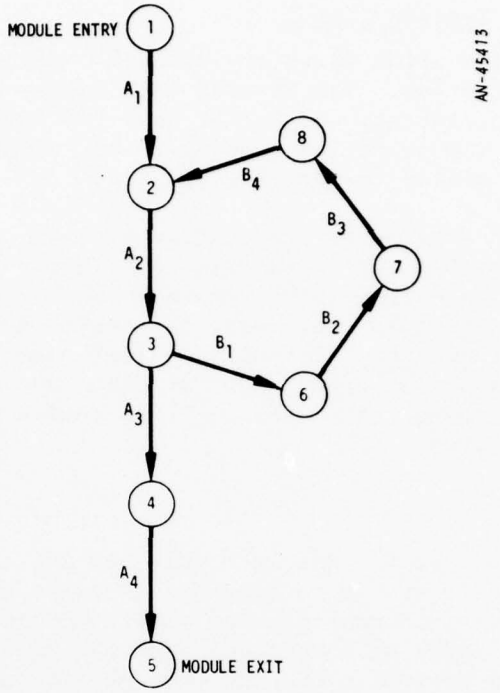


Figure 2.3. Iterative and Non-Iterative Flow Patterns

2.2.2 Reaching Set

The basis for single-module test case data generation is formed from the structural properties of the program as represented by sequences of DD-paths. The set of all DD-paths that connect together to form paths from one designated DD-path to another is called the reaching set. Extracting a particular subset of program text to be analyzed for specific data value settings is accomplished with the reaching set which may or may not include iteration flow patterns. Presumably the DD-path which is the target for a new test case is one which has not been exercised by any of the existing test cases. In the case illustrated (see Fig. 2.3), the members of the DD-path reaching sequence $A_1 \rightarrow A_2 \rightarrow B_1 \rightarrow B_2$ constitute the reaching set which makes the program execute DD-path B_3 from the module entry point. Furthermore it is the only reaching sequence of DD-paths in the reaching set. For more complex programs, the reaching set may include a number of DD-path sequences. Certain of the DD-paths in the reaching set may be essential in that they must be executed in order to reach the designated DD-path. For the example in Fig. 2.3, all the DD-paths in the reaching set (A_1, A_2, B_1, B_2) are essential to reach B_3 from the module entry.

2.2.3 Test Case Data Generation

The test case dataset which forces execution of a particular DD-path can be derived by examining the program text corresponding to a reach sequence which employs the DD-path. While this analysis process is relatively straightforward manually, it is not completely automatable with state-of-the-art techniques.

2.2.4 Selecting the Testing Target

In a typical situation there are several untested DD-paths that need the program tester's attention. The testing methodology does not explicitly decide which of those to concentrate on, but it does provide a program nesting level to help rationalize the selection process. The untested DD-path chosen as the focus of testing is called the testing target.

If all DD-path attributes are otherwise equal, the program tester should choose a target DD-path which is as "high up" in the iteration structure as possible and at the same time as "far down into the code" as possible. By concentrating on a difficult test case, the tester will tend to maximize the amount of collateral testing. Executing a test case designed for some particular DD-path often causes a large number of other DD-paths also to be executed. This phenomenon can reduce the overall effort needed to achieve the 100% testing coverage desired.

2.3 SYSTEM TESTING

The techniques used for single-module testing are employed systematically in achieving full system testing. A large software system is typically composed of subsystems, components, and single modules. Software system testing is performed in terms of the dynamic invocation structure (dynamic organization) of the system, i.e., the tree of intermodule dependencies in which the execution of the software system occurs. Choosing the particular module to which the current testing effort should be directed is based on the dynamic organization, but this choice can also be made by analyzing the single-module coverages already achieved.

2.3.1 System Testing Strategies

Two possible system testing strategies are (1) bottom-up testing and (2) top-down testing. The bottom-up testing strategy first tests single modules at the bottom of the invocation chains, followed by higher-up elements of the software system in their turn. This bottom-up strategy tends to maximize the individually achievable levels of testing at the possible expense of significant problems in constructing the testing environment.

The top-down testing strategy concentrates first on the "topmost" modules of the software system and operates on successively deeper modules within the invocation hierarchy. This method of system testing is likely to produce a large amount of collateral testing. It has the disadvantage, however, that it may be difficult to construct new test case data when the testing target is far from the apex of the invocation hierarchy. This may result from a program's protection of data used by lower-level modules, often a normal and desirable attribute of large-scale software systems.

2.3.2 Coverage and Target Selection

A testing coverage measure consolidates individual module testing coverages into a value applicable to an entire software system. One simple

measure is to consider the software system to be tested as much as its least-tested module, expressed in terms of the exercised percentage of DD-paths. Other measures, which take module and system complexity into account, are also possible.

The testing strategy can use explicit or implicit rules to select the appropriate target for continuing testing efforts. With the simple measure described above the next target is always either (1) the least tested module at the current level within the hierarchy (top down), or (2) the least tested module within the subset of modules currently under analysis or invoked by those modules (bottom up).

2.4 AUTOMATED VERIFICATION SYSTEM DESIGN

An Automated Verification System (AVS) supports the testing methodology by providing various support facilities. The AVS database contains all relevant information about the source text and the structure of the software being analyzed. This information is generated and stored once for each module.

2.4.1 Instrumentation

Individual modules whose source text and related structural information have already been added to the AVS database can be instrumented before execution in a testing environment. The instrumentation techniques are designed to (1) produce program texts which are logically equivalent to the originals, and (2) provide low-overhead invocations to a special instrumentation module which intercepts the program's flow of control as it passes through each DD-path.

2.4.2 Data Collection and Reduction

During tests, the program performs instrumentation invocations that carry with them the name of the module and the number of the DD-path currently being executed. The AVS data collection and reduction facility records these invocations and produces reports which indicate the testing coverage attained for each module.

2.4.3 Test Case Data Generation Assistance

The test case generation assistance facility of the AVS is used when DD-paths within a module are found not to have been executed. Under user-specified commands, the AVS uses the information about the DD-path sequences to generate appropriate reaching sets. The user analyzes the program source text to select the desired DD-path sequence in order to determine appropriate test case data.

2.4.4 Retesting Guidance

In addition to its role in software testing, the AVS provides rudimentary answers to two questions important in maintaining software:

1. If a particular module has been changed, what other modules will have to be retested? Candidates for retesting include modules which are invoked by the changed module, either directly or indirectly, and modules which use the outputs of the changed module.

2. If a series of changes has been made throughout a software system, which modules will have to be retested to restore the system's level of testing? Candidates for retesting are the amalgamated set of modules affected by the changes. The members of this set depend on how the level of testing is defined.

The first question is answered by having the AVS refer to the invocation structure of the software system it has analyzed. The second question is answered by using the instrumentation facility of the AVS: Those modules which have less than the required level of exercise must be re-examined in detail.

2.5 THE IMPACT OF THE AVS-BASED METHODOLOGY

It is difficult to assess the exact impact of widespread use of AVS testing technology because there is only a limited experience basis from which estimates of decreased life-cycle software costs could be made. Based on our initial experience, and admittedly our intuition, we can expect the AVS testing technology to:

- Decrease the overall cost of comprehensively testing software or, equivalently, increase the level of testing coverage achieved for the same cost
- Identify programming constructs which are difficult to handle from a testing point of view and eventually eliminate their use in critical software systems
- Assure that software systems subjected to the AVS-based testing methodology will have a significantly decreased likelihood of failure in actual use, particularly if the software failures could result from lack of comprehensive testing
- Provide a strong technological basis for continued development of the methodologies and the tools which support them.

In the long run, the benefits of AVS technology will occur through (1) increased understanding of the general problem of software testing, and (2) development of better automated tools.

2.6 SUMMARY

The main points made in this section are the following:

- Among a variety of approaches to improving software quality, systematic program testing offers near-term benefits not available with other techniques.

- The most difficult problem in testing is the construction of test case data which exercises previously unexercised parts of programs.
- Detailed analysis of a program's control structure can be used to construct test case data.
- The methodology for single-module testing can be extended to system (multi-module) testing.
- The simplest measures of testing coverage, if achieved, have a positive effect on installed software system quality.
- Systematic software testing can be supported by the facilities of an Automated Verification System (AVS).
- Systematizing the program testing process can reduce software quality enhancement costs appreciably.

These topics are explored in depth in Ref. 2, Methodology for Comprehensive Software Testing.

3 OVERVIEW OF AVS CAPABILITIES

This section is an overview of the capabilities provided by JAVS, an AVS implementation for programs written in JOVIAL/J3. These capabilities typify those implemented for other programming languages such as FORTRAN[10] and PASCAL.[12]

JAVS was developed as a tool to aid JOVIAL software developers and testers in determining the extent to which their programs have been tested and to assist in deriving additional test cases to verify the software. Up to now, testing has been without an orderly approach and without accurate means to determine exactly what portions of code have been exercised. JAVS is an automated tool for measuring the effectiveness of test data in terms of program structure, and this report provides a testing approach to be used with JAVS.

3.1 CAPABILITIES OF JAVS

JAVS will analyze as many as 250 invokable modules and an unlimited number of JOVIAL statements in a single processing job. A module is a JOVIAL main program, CLOSE, or PROC. A START-TERM sequence is a unit of source text, containing one or more modules, which is separately compilable (for the JOCIT JOVIAL compiler). Programs containing more than 250 modules must be partitioned into "components," which are groups of START-TERM sequences. If partitioning a program is necessary, functionally similar START-TERM sequences should be kept in the same component for JAVS processing.

The role of JAVS, as a testing tool, is to assure that the software has achieved a measurable level of exercise. JAVS provides execution coverage reports showing which modules, DD-paths, and statements have been exercised. When test cases are input which achieve a high level of DD-path coverage and which match the requirements stated in a functional specification, the tester can be assured of comprehensive verification.

The dynamic behavior of the program can be studied by requesting JAVS tracing reports. These traces show the invocations and returns of all modules executed during the test. At user option, the tracing can be performed at the DD-path level to determine the dynamic behavior of the program while it is processing the data. In addition, the user can trace "important" events, such as overlay link loading, by invoking one of the JAVS data collection routines.

Single-module testing is oriented toward exercising all DD-paths within the module. The basis for single-module test case data generation is formed from the reaching set (see Sec. 2.2.2). When a testing target DD-path (or set of targets) has been identified from the coverage reports, the user can get assistance from JAVS in determining the reaching set to the target. Armed with the "reaching set" report, the user can spot key variables whose values affect the flow through the program path and locate all instances of the variables in the system-wide cross reference.

JAVS supports top-down and bottom-up system testing strategies by producing reports which show the modules' interaction in terms of invocation description and calling trees. JAVS provides matrices showing interaction of modules within the system examined by JAVS and invocations of modules external to the system.

JAVS uses a data base to store information about the test program. The availability and management of this information form the basis for a variety of services, in addition to the primary task of testing assistance. Computer program documentation, debugging through JAVS computation directives, and reports useful for code optimization are the major side benefits of JAVS.

Computer documentation requirements for the Air Force typically specify flow charts and lists of program variables and constants. In the JAVS development and implementation contracts, these requirements were replaced by specifying certain JAVS reports; i.e., self-documentation. It was found that the module listings (enhanced by indentation and identification of decision points), module control flow pictures, module invocation reports (showing formal and actual parameter lists), module interdependence reports, and a cross reference report for each JAVS component are more meaningful documentation and are generated automatically by JAVS.

Software development can be assisted by using JAVS to document and test the system as it is built. To aid in data flow analysis and bounds checking, JAVS offers computation directives. The directives are a special form of JOVIAL comment, recognized by JAVS and expanded into executable code (using the JOVIAL monitor statement) during the instrumentation phase. The user can check logic expressions with an ASSERT directive, check boundaries of selected variables with an EXPECT directive, and turn on and off the standard monitor tracing with TRACE and OFFTRACE directives.

Code optimization is aided by the post-test reports which show the number of times each statement is executed and the execution time (in central processor milliseconds) spent in the modules. Modules which are never called and should be removed are listed in another JAVS report.

3.2 LIMITATIONS OF JAVS

Testing coverage results indicate what parts of the program were executed. It is up to the user to determine if the program's output is reasonable. One of the JAVS post-test analysis reports lists the execution coverage during the test run in terms of the percentage of decision outways taken. A good standard for the level of testing of a program is to exercise every decision outway (that is every DD-path) at least once. This level of testing is more rigorous than simply testing every program statement at least once. However, it should be emphasized that certain combinations of DD-paths may cause errors which are not detected in merely executing each outway one time.

3.3 ORGANIZATION OF JAVS

JAVS reads the user's JOVIAL program as data and performs syntactical, structural, and instrumentation analyses on the source code. JAVS communicates with the user through a command language and utilizes a data base to store the information about the program. The user is provided with an instrumented file of the selected program modules with which the user supplies test data for execution. The execution results are written to a file for use by JAVS's post-test analyzer which issues execution tracing and coverage reports.

Six functional processes, in addition to execution with test data, make up the substance of software validation provided by JAVS. The organization of JAVS is defined by these six tasks. To reduce the burden of the user, JAVS exists as an overlay program at RADC with a macro command language supplementing a large, versatile standard command language. The processing steps and their basic functions are listed below:

BASIC, Source Text Analysis: Source text input, lexical analysis, and initial source library creation

STRUCTURAL, Structural Analysis: Structural analysis and execution path identification; library update with structure and path information

INSTRUMENT, Module Instrumentation: Program instrumentation for path coverage analysis and program performance directed by user; library update with probe test instrumentation

ASSIST, Module Testing Assistance and Segment Analysis: Testing assistance for improved program coverage

DEPENDENCE, Retesting Guidance and Analysis: Retesting requirements analysis for changed modules

Test Execution: Execution of instrumented code and analysis of directed program performance

ANALYZER, Test Effectiveness Measurement: Detailed analysis of program path coverage; execution traces and summary statistics

These steps need not be performed in the above order. Other orders may be preferable at times.

Table 3.1 shows the relationship between the macro command, standard command, and processing task.

An overview of how JAVS is used in the testing process is shown in Fig. 3.1.

TABLE 3.1

RELATIONSHIP BETWEEN COMMANDS AND TASKS

Macro Command Keyword	Standard Command Keyword	Task
BUILD LIBRARY	BASIC	Syntax analysis
	STRUCTURAL	Structural analysis
PROBE	INSTRUMENT	Structural and computation instrumentation
DOCUMENT	ASSIST	Module and intermodule reports
	PRINT	
	DEPENDENCE	
TEST	ANALYZER	Post-test coverage and trace analysis

3.4 SUMMARY

The main points of this section are the following:

- JAVS offers immediate benefits in systematically testing, documenting, and maintaining JOVIAL software.
- JAVS offers assistance in retesting software, but test case generation and functional analysis of the program's output are primarily tasks of the tester.
- The processes accomplished by JAVS are: syntax and structural analyses, instrumentation, retesting assistance, and module interdependence and post-test analyses.
- The user interacts with JAVS via a command language.

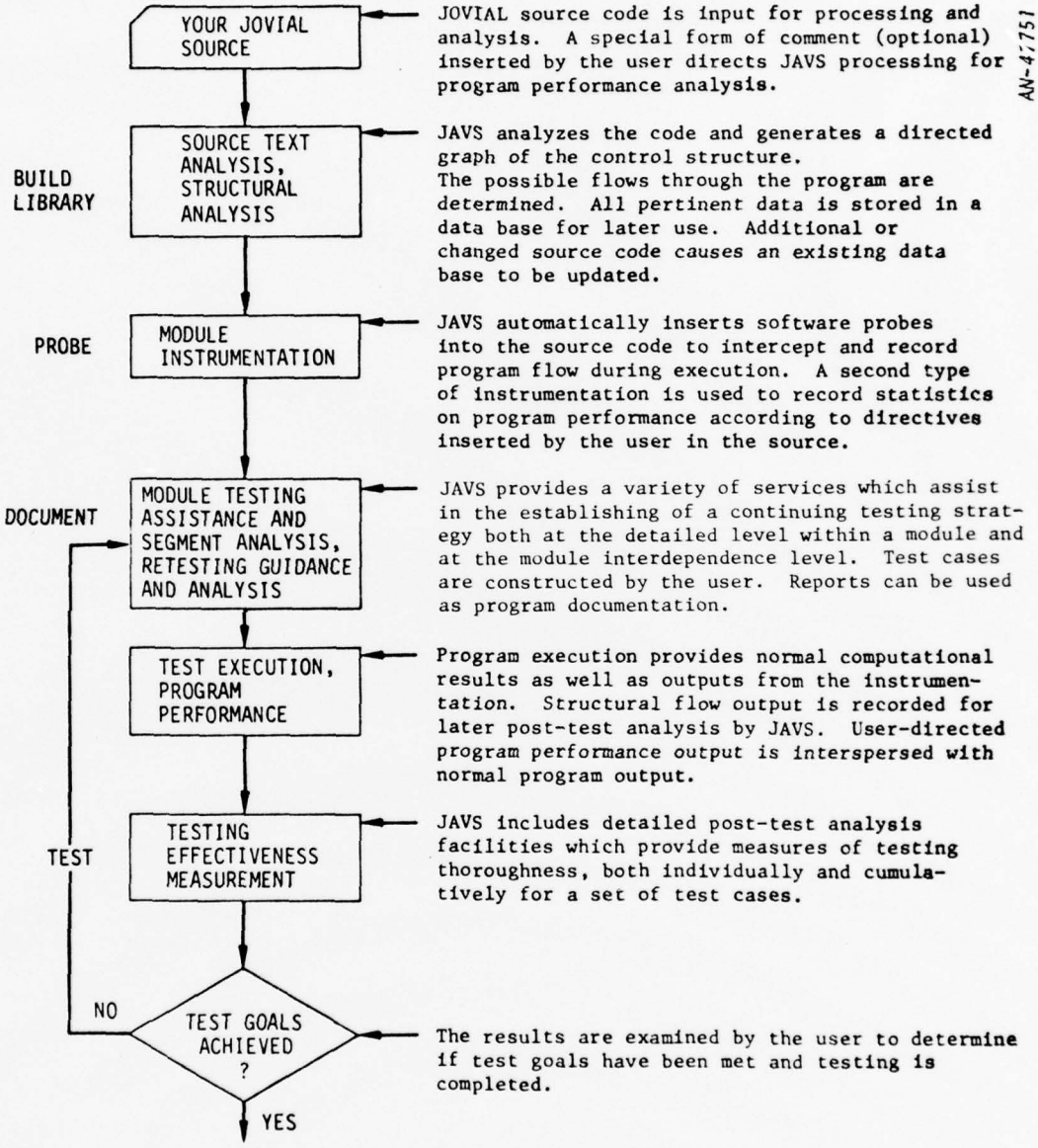


Figure 3.1. Overview of JAVS in the Testing Process

4 REVIEW OF JAVS TESTING EXPERIENCE

To date, two formal test activities which illustrate aspects of the testing methodology have been conducted using JAVS. These test activities are the JAVS software acceptance tests (part of the initial contract) and the JAVS evaluation tests (part of the current contract). Detailed descriptions of these tests and the results appear in Refs. 7, 9, and 13.

The acceptance tests were conducted using as software test objects the following:

- TESTALL. Implementation and installation tests specifically designed to stress functional features of individual JAVS components
- BLSTIC. A representative JOVIAL application program
- JAVS software itself

The evaluation tests used the COMPOSE component of SAC's Force Management Information System (FMIS). For both series of tests, the test objects range in size from small to large (e.g., BLSTIC is 366 statements and COMPOSE is 38,734 statements excluding comments).

The goals of the acceptance tests were to:

1. Demonstrate standard usage of JAVS in an operational environment
2. Execute 85% of JAVS source code

The goals of the evaluation tests were to:

1. Identify both strong and weak areas of JAVS when used to process large software systems.
2. Verify that all portions of JAVS are operating properly and that current documentation is accurate. Deficiencies found in JAVS during the testing process were to be corrected if the resources required are within the limits of the effort.
3. Identify and recommend areas of JAVS which could be subject to future enhancement.
4. Document the results of the testing process for future reference and study.
5. Establish recommended procedures for using JAVS to test and verify large software systems.

As a secondary benefit, portions of the FMIS software were tested and documented.

The remainder of this section discusses the application of the testing methodology to each test activity.

4.1 ACCEPTANCE TESTS

The approach taken in JAVS acceptance tests was to group the tests into functional, system-wide,* and self-test categories. Collateral testing was used whenever possible to minimize the number of tests within each category. Although only the system-wide and self-test categories are pertinent to the application of the methodology, some of the JAVS functional testing was satisfied by the system-wide tests and vice versa.

TESTALL, the JOVIAL program used as data for JAVS for the functional tests, consists of three START-TERM sequences--a COMPOOL and two main programs--and a number of procedures and CLOSEs. It is not an application program and has no other purpose than to exercise JAVS.

For the system-wide testing, an engineering application program, supplied by RADC, called BLSTIC was used. It contains a main program and several internal procedures. As part of the acceptance tests, a typical application of JAVS to program testing was posed as a problem to solve: use JAVS on BLSTIC to obtain full DD-path coverage and, using JAVS, explain all unreachable source code in the program.

For the first Test Execution, test data supplied with BLSTIC was used. Subsequent executions used data generated with the assistance of JAVS retesting guidance.

Two files were used as data for the JAVS self-tests: TESTALL was used on the entire JAVS system, and TSTMOR was used on several large modules in the JAVS-2 component, to achieve the required 85% statement coverage. TSTMOR is TESTALL plus one more START-TERM sequence, containing many "peculiar" JOVIAL statements to exercise some of the lesser-known features of the language which JAVS-2 seeks during its source text analysis.

To demonstrate coverage of JAVS software, a technique (self-test) of using JAVS to analyze its own software as a test object was adopted. JAVS consists of twelve major components, each of which has its own COMPOOL and a number of executable modules; several components have nearly 100 modules apiece. Since the JAVS software is sizable (approximately 30,000 statements), the approach to testing was to test each component separately, using as the data for Test Execution the TESTALL source code.

4.1.1 Test Object BLSTIC

The testing of BLSTIC demonstrated the capabilities of JAVS as a testing tool on an existing engineering application program. Although BLSTIC is neither large nor complex, it was well-suited for a demonstration: its source text was of a manageable size, initial test data was available, the program had not been previously analyzed by JAVS, and the tester had no prior knowledge of the program.

---

*In this context system-wide tests mean the application of JAVS to a complete program for the purposes of comprehensively testing that program.

Since one purpose of the test was to demonstrate all JAVS functions, all the JAVS processing steps were executed for BLSTIC and every applicable JAVS command was used. When all JAVS functions are executed, the volume and variety of reports produced is large; under more normal conditions a tester needs much less than the complete set to meet particular test objectives. The properties of the program being tested (e.g., data organization, modularity, complexity of control structure, test data) as well as knowledge about the program have some bearing on the analyst's testing strategy and choice of JAVS reports.

Starting the Test. The initial step in testing BLSTIC was to process it through BASIC, STRUCTURAL, INSTRUMENT, Test Execution, and ANALYZER, using as test data the single test case supplied by RADC. ASSIST and DEPENDENCE reports were also generated.

Selecting a Target Module. The SUMMARY and NOTHIT coverage reports[5] from ANALYZER were examined to identify modules with poor coverage. The summary report (Fig. 4.1) shows that the two modules with the most DD-paths (DECMAL and FLTOUT) achieved the lowest percentage coverage with the single test case. One approach, which often maximizes collateral testing and minimizes the analyst's time in deriving new test cases, is to concentrate on the largest modules with the poorest coverage. In this instance, DECMAL and FLTOUT have almost the same number of DD-paths (Fig. 4.1) and achieved about the same coverage (Fig. 4.2). DECMAL was selected as the testing target, because of the deep control nesting levels of its unexercised DD-paths. This is determined by comparing the list of unexercised paths to the control-structure nesting level shown on one of several reports: the module listing (Figs. 4.3 and 4.4), the DD-path definition listing (Fig. 4.5), or the DD-path picture report (Fig. 4.6).

Selecting a Target DD-Path. Having selected a target module, the next step is to select one or more target DD-paths which have not yet been executed. Testing a DD-path which is on a high iteration level or deep in the control nesting structure results in good collateral testing coverage. Figure 4.2 listed the DD-paths not executed during the first test case. In the retesting target module DECMAL, the list begins with 2, 4, 5, 6, 7.... The DD-path definition report (Fig. 4.5) shows DD-path 6 as being nested at the fifth control level. Thus, if DD-path 6 is executed, so are paths 2 and 4.

Preparing a New Test. A display of the iterative reaching set for DD-path 6 (Fig. 4.7) shows the set of executable statements which may be executed in order to reach the target DD-path. At this point, the testing analyst knows what statements lead up to the untested DD-path, but it may not be obvious what input data changes to make. Backtracking the statements in the reaching set shows that "I" must be less than 11, one byte of "ARG" must be a blank, 0(20), and the byte of "ARG" before the blank must be a decimal point, 0(33). Statement 18 states that ARG = INPUT1, so a new test case for the single module would be to supply an appropriate value for INPUT1 to module DECMAL (for example "five, decimal point, blank") and execute the single module with a dummy main program.

System-wide testing, however, requires the analyst to test the entire system at once, or by components which are interrelated. JAVS offers assistance in tracking down which of the input parameters to change for a new test case. Having tracked INPUT1 to its declaration as a formal input parameter for procedure DECMAL (statement 1 in Fig. 4.3 or Fig. 4.5), the Library Cross Reference Listing (Fig. 4.8) can be used to locate where it is defined and set. The Module Invocation Space report for DECMAL (Fig. 4.9) shows that DECMAL was called by module INFO at statement 7. If further tracking is necessary, the Module Invocation Bands report (Fig. 4.10) shows that INFO, which called DECMAL, is called by BLSTIC, the main program.

Manual backtracking to generate new test data is effective for applications like BLSTIC which have few levels of module invocation and few transformations between the input of test data and the test target.

By deriving test cases several at a time, rerunning Test Execution and ANALYZER, and retesting, satisfactory testing results on BLSTIC were achieved. The report in Fig. 4.11 shows that in 11 test cases all modules except DECMAL and FLTOUT received full coverage. (Modules BLSTIC and CONVRT each had DD-paths not exercised during each individual test case, but cumulatively all DD-paths were hit.) The report shows which DD-paths have not been exercised.

Locating Dead Code. JAVS structural analysis identifies any source code which cannot be reached solely because of the control structure of the program (structurally unreachable code). Identifying logically unreachable code requires analysis of both the control structure and the values of the data. This type of analysis is called data flow analysis,[12] a capability not yet available in JAVS.

For applications like BLSTIC, however, JAVS does offer some assistance in locating logically dead code. Figure 4.11 shows the DD-paths not executed during any of the 11 test cases. Only modules FLTOUT and DECMAL have DD-paths unhit. Figures 4.12 and 4.13 are the DD-path coverage listings for the two modules, DECMAL and FLTOUT. Analysis of the module statements in DECMAL (Fig. 4.3, statement 36) (or using a reaching set listing) shows that DD-path 11 can never be executed, since PLACE is always set to 12. Thus that particular IF test (statement 39) could be removed with no change in the output. In module FLTOUT, DD-path 16 can be exercised, but only if the appropriate combination of input data yields an exponent of exactly 10 after a considerable number of calculations (see Fig. 4.4, statements 69-71). Continuing the analysis, it can be seen that all other unexercised DD-paths can never be exercised, so the logically unreachable code can be removed from the module.

Summary. The testing experience with BLSTIC may be summarized as follows:

- JAVS reports provided the tester, who had no prior knowledge of BLSTIC, with comprehensive information about the organization of the program into modules and their relations, the control structure of the individual modules, and the use of data in the program.

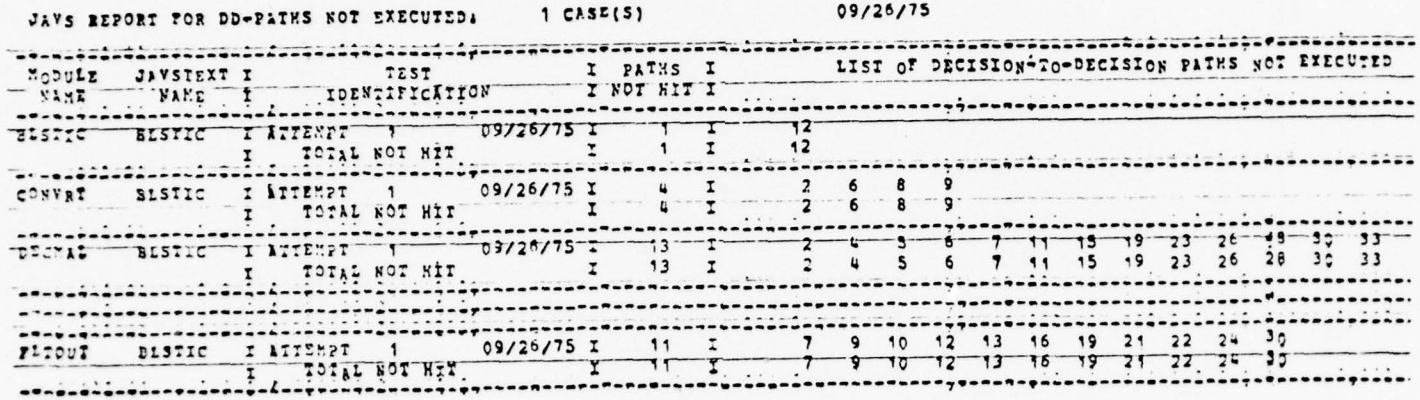
- After testing with supplied test data, construction of new tests was a straightforward process with the testing assistance capabilities of JAVS.
- By using JAVS reports in a systematic way the tester was able to meet the testing objective for BLSTIC: namely, to obtain full DD-path coverage and explain all unreachable source code in the program.
- The testing techniques used a combination of automated analysis and manual backtracking which may be difficult to apply to system-wide testing of large, complex programs.

1 CASE(S) 09/26/75

ATTEMPT 1 09/26/75

MODULE NAME	JAVSTEXT NAME	NUMBER OF D-D PATHS	SUMMARY - THIS TEST: NUMBER OF INVOCATIONS	D-D PATHS TRAVERSED	PER CENT COVERAGE	CUMULATIVE SUMMARY: NUMBER OF TESTS	NUMBER OF INVOCATIONS	D-D PATHS TRAVERSED	PER CENT COVERAGE
BLSTIC	BLSTIC	13	1	12	92	1	1	12	92
CONVRT	BLSTIC	13	8	9	69	1	8	9	69
DECMAL	BLSTIC	33	4	20	60	1	4	20	60
LOAD	BLSTIC	1	10	1	100	1	10	1	100
INFO	BLSTIC	1	4	1	100	1	4	1	100
PUTOUT	BLSTIC	31	10	20	64	1	10	20	64
*****	ALL *****	92		63	68			63	68

Figure 4.1. Single Test Case Summary Report of DD-Path Coverage



JAVS REPORT FOR DD-PATHS NOT EXECUTED: 1 CASE(S) 09/26/75

MODULE NAME	JAVSTEXT NAME	I	TEST IDENTIFICATION		I	PATHS NOT HIT	I	LIST OF DECISION-TO-DECISION PATHS NOT EXECUTED
BLSTIC	BLSTIC	I	ATTEMPT	1 09/26/75	I	1	I	12
		I	TOTAL NOT HIT		I	1	I	12
CONVRT	BLSTIC	I	ATTEMPT	1 09/26/75	I	4	I	2 6 8 9
		I	TOTAL NOT HIT		I	4	I	2 6 8 9
DECMAT	BLSTIC	I	ATTEMPT	1 09/26/75	I	13	I	2 4 5 6 7 11 15 19 23 26 28 30 33
		I	TOTAL NOT HIT		I	13	I	2 4 5 6 7 11 15 19 23 26 28 30 33
FLTOUT	BLSTIC	I	ATTEMPT	1 09/26/75	I	11	I	7 9 10 12 13 16 19 21 22 24 30
		I	TOTAL NOT HIT		I	11	I	7 9 10 12 13 16 19 21 22 24 30

Figure 4.2. Unexercised DD-Paths

```
MODULE STATEMENT LISTING

MODULE <DECMAL  >, JAVSTEXT <BLSTIC  >, PARENT MODULE <BLSTIC  >

 NO.  LVL            STATEMENT                                                    DO-PATHS                    CONTROL
-------------------------------------------------------------------------------------------------------------------
   1 ( 0)            PROC DECMAL ( INPUT1 ) $                                   (  1)
   2 ( 0)            ITEM DECMAL F $
   3 ( 0)            ITEM INPUT1 H 12 $
   4 ( 0)            ITEM ARG H 12 $
   5 ( 0)            ITEM PLACE I 36 S $
   6 ( 0)            ITEM INTGRH H 12 P 12H(            ) $
   7 ( 0)            ITEM FRCTNH H 12 P 12H(            ) $
   8 ( 0)            ITEM INTGR I 36 S $
   9 ( 0)            ITEM FRCTN I 36 S $
  10 ( 0)            ITEM INTGRF F $
  11 ( 0)            ITEM FRCTNF F $
  12 ( 0)            ITEM COUNT1 I 36 S $
  13 ( 0)            ITEM COUNT2 I 36 S $
  14 ( 0)            ITEM KK B P 0 $
  15 ( 0)            ITEM PP B P 0 $
  16 ( 1)              BEGIN
  17 ( 1)            ## DECMAL ##
  18 ( 1)              ARG = INPUT1 $
  19 ( 1)              KK = 0 $
  20 ( 1)              PP = 0 $
  21 ( 1)              INTGRH = 12H(            ) $
  22 ( 1)              FRCTNH = 12H(            ) $
  23 ( 1)              FOR I = 0 , 1 , 11 $                                                                    FOR3
  24 ( 2)                BEGIN
  25 ( 2)            ## I ##
  26 ( 2)                IF BYTE ($ I $) ( ARG ) EQ O(33) $                       (  2-   3)                    IF
  27 ( 3)                  BEGIN
  28 ( 3)                  PLACE = I $
  29 ( 3)                  IF BYTE ($ I + 1 $) ( ARG ) EQ O(20) $                 (  4-   5)                    IF
  30 ( 4)                    BEGIN
  31 ( 4)                    FOR G = I , 1 , 11 $                                                              FOR3
  32 ( 5)                      BYTE ($ G $) ( ARG ) = O(00) $                     (  6-   7)
  33 ( 4)                    END
  34 ( 3)                  GOTO S1 $                                                                           ------>
  35 ( 3)                  END
  36 ( 2)                PLACE = 12 $
  37 ( 2)                END                                                      (  8-   9)
  38 ( 1)            ## I ##
  39 ( 1)              IF PLACE EQ 12 $                                           ( 10-  11)                    IF
  40 ( 2)                BEGIN
  41 ( 2)                FOR K = 0 , 1 , 11 $                                                                  FOR3
  42 ( 3)                  BEGIN
  43 ( 3)                  IF BYTE ($ K $) ( ARG ) EQ O(20) $                     ( 12-  13)                    IF
  44 ( 4)                    BEGIN
  45 ( 4)                    IF PP $                                              ( 14-  15)                    IF
  46 ( 5)                      BEGIN
  47 ( 5)                      PLACE = K $
  48 ( 5)                      FOR J = K , 1 , 11 $                                                            FOR3
  49 ( 6)                        BYTE ($ J $) ( ARG ) = O(00) $                   ( 16-  17)
  50 ( 5)                      GOTO S1 $                                                                       ------>
  51 ( 5)                      END
  52 ( 4)                    GOTO R1 $                                                                         ------>
  53 ( 4)                    END
  54 ( 3)                  PP = 1 $
  55 ( 3)          R1.                                                          ( 18-  19)                    <------
                     END
```

Figure 4.3. Module Listing for DECMAL

```
55 ( 2)                 DECMAL = CONVRT ( ARG ) $
57 ( 2)                 GOTO S10 $                                                          ------>
58 ( 2)                 END
59 ( 1)         S1.                                                                         FOR3<------
                    FOR I = 0 , 1 , 10 $
60 ( 2)                 BEGIN
61 ( 2)               ## I ##
62 ( 2)                 IF BYTE ($ I $) ( ARG ) EQ O'00' $                  ( 20-  21)      IF
63 ( 3)                   GOTO S2 $                                                         ------>
64 ( 2)                 IF BYTE ($ I $) ( ARG ) NQ O'52' $                  ( 22-  23)      IF
65 ( 3)                   GOTO S2 $                                                         ------>
66 ( 2)                 KK = 1 $
67 ( 2)                 BYTE ($ I +1) ( ARG ) = O'00' $
68 ( 2)                 GOTO S3 $                                                           ------>
69 ( 2)         S2.                                                 ( 24-  25)              <------
                        END
70 ( 1)               ## I ##
71 ( 1)         S3.                                                 ( 26-  27)      IF      <------
                    IF PLACE EQ 0 $
72 ( 2)                 BEGIN
73 ( 2)                 INTGRH = 12H(000000000000) $
74 ( 2)                 GOTO RX $                                                           ------>
75 ( 2)                 END
76 ( 1)             BYTE ($ 12 - PLACE , PLACE $) ( INTGRH ) = BYTE ($ 0 , PLACE $) (
                ARG ) $
77 ( 1)             IF PLACE EQ 11 $                                ( 28-  29)      IF
78 ( 2)               BEGIN
79 ( 2)               DECMAL = CONVRT ( INTGRH ) $
80 ( 2)               GOTO S9 $                                                         ------>
81 ( 2)               END                                                               <------
82 ( 1)         RX.
                    BYTE ($ PLACE + 1 , 11 - PLACE $) ( FRCTNH ) = BYTE ($ PLACE + 1
                , 11 - PLACE $) ( ARG ) $
83 ( 1)             IF PLACE EQ 0 $                                 ( 30-  31)      IF
84 ( 2)               BEGIN
85 ( 2)               INTGR = 0 $
86 ( 2)               GOTO RY $                                                         ------>
87 ( 2)               END
88 ( 1)             INTGR = CONVRT ( INTGRH ) $                                         <------
89 ( 1)         RY.
                    FRCTN = CONVRT ( FRCTNH ) $
90 ( 1)             INTGRF = INTGR $
91 ( 1)             FRCTNF = FRCTN $
92 ( 1)             FRCTNF = FRCTNF / ( 10 ** ( 11 - PLACE ) ) $
93 ( 1)             DECMAL = INTGRF + FRCTNF $
94 ( 1)         S9.                                                 ( 32-  33)      IF      <------
                    IF NOT KK $
95 ( 2)               GOTO S10 $                                                        ------>
96 ( 1)             DECMAL = - DECMAL $                                                 <------
97 ( 1)         S10.
                    END
```

Figure 4.3. (Cont.)

```
MODULE STATEMENT LISTING

MODULE <FLTOUT  >, JOVSTEXT <BLSTIC  >, PARENT MODULE <BLSTIC  >

NO.  LVL          STATEMENT                                                    DO-PATHS        CONTROL
---------------------------------------------------------------------------------------------------------
  1 ( 0)          PROC FLTOUT ( INPUT1 = SEQNM ) $                             (  1)
  2 ( 0)          ITEM OUTPT I 36 S $
  3 ( 0)          ITEM SEQNM H 16 $
  4 ( 0)          ITEM DEC F $
  5 ( 0)          ITEM YY F $
  6 ( 0)          ITEM CON A 35 U $
  7 ( 0)          ITEM TEM A 35 U $
  8 ( 0)          ITEM J2 I 36 S $
  9 ( 0)          ITEM J1 I 36 S $
 10 ( 0)          TABLE PRETAB R 16 S $
 11 ( 1)            BEGIN
 12 ( 1)            ITEM PRE A 35 U $
 13 ( 1)            END
 14 ( 0)          ITEM INPUT1 F $
 15 ( 0)          ITEM BITE 4 1 $
 16 ( 0)          ITEM LITTE I 6 S $
 17 ( 0)          OVERLAY LITTE = BITE $
 18 ( 1)            BEGIN
 19 ( 1)          ** FLTOUT **
 20 ( 1)            CON = 1 $
 21 ( 1)            OUTPT = INPUT1 $
 22 ( 1)            BYTE ($ 0 $) ( SEQNM ) = O(20) $
 23 ( 1)            BYTE ($ 2 $) ( SEQNM ) = O(33) $
 24 ( 1)            FOR I = 3 , 1 , 15 $                                                        FOR3
 25 ( 2)              BYTE ($ I $) ( SEQNM ) = O(00) $                           (  2-  4)
 26 ( 1)            BYTE ($ 11 $) ( SEQNM ) = O(25) $
 27 ( 1)            IF OUTPT EQ 0 $                                            (  4-  5)        IF
 28 ( 2)              BEGIN
 29 ( 2)              BYTE ($ 1 $) ( SEQNM ) = O(60) $
 30 ( 2)              BYTE ($ 12 $) ( SEQNM ) = O(60) $
 31 ( 2)              GOTO J26 $                                                                  ------>
 32 ( 2)              END
 33 ( 1)            IF OUTPT GR 0 $                                            (  6-  7)        IF
 34 ( 2)              BEGIN
 35 ( 2)              BYTE ($ 1 $) ( SEQNM ) = O(60) $
 36 ( 2)              YY = OUTPT $
 37 ( 2)              GOTO J1 $                                                                   ------>
 38 ( 2)              END
 39 ( 1)            BIT ($ 27 , 9 $) ( OUTPT ) = 777777777 - BIT ($ 27 , 9 $) ( OUTPT
                  ) $
 40 ( 1)            YY = - OUTPT $
 41 ( 1)            BYTE ($ 1 $) ( SEQNM ) = O(52) $
 42 ( 1)          J1.                                                          (  8-  9)        IF <------
 43 ( 2)            IF YY GQ 1.E-001 $
 44 ( 2)              BEGIN
 45 ( 2)              BYTE ($ 12 $) ( SEQNM ) = O(60) $
 46 ( 2)              IF ( YY LS 1.E+000 ) $                                     ( 10- 11)        IF
 47 ( 3)                BEGIN
 48 ( 3)                CON = 0 $
 49 ( 3)                GOTO J20 $                                                                ------>
 50 ( 3)                END
 51 ( 2)              GOTO J10 $                                                                  ------>
 52 ( 2)              END
 53 ( 1)            BYTE ($ 12 $) ( SEQNM ) = O(52) $
 54 ( 1)            DEC = 1.E-002 $
 55 ( 1)            FOR I = 0 , 1 $                                                             FOR2
```

Figure 4.4. Module Listing for FLTOUT

```
55 ( 2)                 BEGIN
56 ( 2)                 IF YY GQ DEC $                                  ( 12- 13)       IF
57 ( 3)                   GOTO J12 $                                                      ------>
58 ( 2)                 CON = CON + 1 $
59 ( 2)                 DEC = DEC * 1.E-001 $
60 ( 2)                 END
61 ( 1)           J19.                                                                  <------
                  DEC = 10.E+000 $
                  FOR I = 0 , 1 $                                                       FOR2
62 ( 2)                 BEGIN
63 ( 2)                 IF YY LS DEC $                                  ( 14- 15)       IF
64 ( 3)                   GOTO J12 $                                                    ------>
65 ( 2)                 CON = CON + 1 $
66 ( 2)                 DEC = DEC * 100.E-001 $
67 ( 2)                 END
68 ( 1)           J12.                                                                  <------
                  PRE ($ 15 $) = CON / 10 $
                  TEM = PRE ($ 15 $) * 10 $
                  IF CON - TEM EQ 0 $                             ( 16- 17)       IF
69 ( 2)                   GOTO J14 $                                                    ------>
70 ( 1)                 BIT ($ 90 , 6 $) ( SEQNM ) = CON - TEM $
71 ( 1)           J14.                                                                  <------
                  PRE ($ 14 $) = PRE ($ 15 $) / 10 $
                  TEM = PRE ($ 14 $) * 10 $
72 ( 1)                 IF PRE ($ 15 $) - TEM EQ 0 $                    ( 18- 19)       IF
73 ( 2)                   GOTO J15 $                                                    ------>
74 ( 1)                 BIT ($ 84 , 6 $) ( SEQNM ) = PRE ($ 15 $) - TEM $
75 ( 1)           J15.                                                                  <------
                  IF CON / 100 EQ 0 $                             ( 20- 21)       IF
76 ( 2)                   GOTO J20 $                                                    ------>
77 ( 1)                 BIT ($ 78 , 6 $) ( SEQNM ) = CON / 100 $                        ------>
78 ( 1)                 GOTO J20 $                                                      <------
79 ( 1)           J20.                                                  ( 22- 23)       IF
                  IF YY LS 1.E-001 $
80 ( 2)                 BEGIN
81 ( 2)                 PRE ($ 11 $) = YY * 10.E+000 ** ( 8 + CON ) $
82 ( 2)                 GOTO J22 $                                                      ------>
83 ( 2)                 END
                  IF YY GQ 1000000000.E+000 $                     ( 24- 25)       IF
84 ( 2)                 BEGIN
85 ( 2)                 PRE ($ 11 $) = YY / 10 ** ( CON - 8 ) $
86 ( 2)                 GOTO J22 $                                                      ------>
87 ( 2)                 END
88 ( 1)                 PRE ($ 11 $) = YY * 10.E+000 ** ( 8 - CON ) $
89 ( 1)           J22.                                                                  FOR3<------
                  FOR I = 10 , - 1 , 3 $
90 ( 2)                 BEGIN
91 ( 2)                 PRE ($ I $) = PRE ($ I + 1 $) / 10 $
92 ( 2)                 TEM = PRE ($ I $) * 10 $
93 ( 2)                 IF PRE ($ I + 1 $) - TEM EQ 0 $                 ( 26- 27)       IF
94 ( 3)                   GOTO J24 $                                                    ------>
95 ( 2)                 J2 = I * 6 $
96 ( 2)                 J3 = PRE ($ I + 1 $) - TEM $
97 ( 2)                 BIT ($ J2 , 6 $) ( SEQNM ) = J3 $
98 ( 2)           J24.                                                  ( 28- 29)       <------
                        END
99 ( 1)           J26.                                                  ( 30- 31)       IF <------
                  IF INPUT1 LS 0 $
100 ( 2)                BYTE ($ 1 $) ( SEQNM ) = O(52) $
101 ( 1)                BYTE ($ 0 $) ( SEQNM ) = O(20) $
102 ( 1)                BYTE ($ 2 $) ( SEQNM ) = O(33) $
103 ( 1)                IF BYTE ($ 13 $) ( SEQNM ) EQ O(20) $           ( 32- 33)       IF
104 ( 2)                  BYTE ($ 13 $) ( SEQNM ) = O(00) $
105 ( 1)                FOR K = 3 , 1 , 10 $                                            FOR3
106 ( 2)                  BEGIN
107 ( 2)                  IF NOT ( BYTE ($ K $) ( SEQNM ) EQ 1H(0) ) $  ( 34- 35)       IF
108 ( 3)                    BEGIN
109 ( 3)                    BYTE ($ 0 $) ( BITE ) = BYTE ($ K $) ( SEQNM ) $
110 ( 3)                    IBITE = IBITE + O(00) $
111 ( 3)                    BYTE ($ K $) ( SEQNM ) = BYTE ($ 0 $) ( BITE ) $
112 ( 3)                    END
113 ( 2)                  END                                           ( 36- 37)
114 ( 1)                FOR K = 14 , 1 , 15 $                                           FOR3
115 ( 2)                  BEGIN
116 ( 2)                  IF NOT ( BYTE ($ K $) ( SEQNM ) EQ 1H(0) ) $  ( 38- 39)       IF
117 ( 3)                    BEGIN
118 ( 3)                    BYTE ($ 0 $) ( BITE ) = BYTE ($ K $) ( SEQNM ) $
119 ( 3)                    IBITE = IBITE + O(00) $
120 ( 3)                    BYTE ($ K $) ( SEQNM ) = BYTE ($ 0 $) ( BITE ) $
121 ( 3)                    END
122 ( 2)                  END                                           ( 40- 41)
123 ( 1)                RETURN $                                                        ------>
124 ( 1)                END
```

Figure 4.4. (Cont.)

```
MODULE DD-PATH DEFINITION LISTING

MODULE <DECMAL  >, JAVSTEXT <RLSTIC  >, PARENT MODULE <RLSTIC  >

 NO.  LVL          STATEMENT                                   DD-PATHS GENERATED
-------------------------------------------------------------------------------------------
   1 ( 0)          PROC DECMAL ( INPUT1 ) $
                                                               ** DD-PATH   1 IS PROCEDURE ENTRY
                   . . . .
  23 ( 1)          FOR I = 0 , 1 , 11 $
  24 ( 2)            BEGIN
                     . . . .
  25 ( 2)            IF BYTE ($ I $) ( ARG ) EQ O(35) $
                                                               ** DD-PATH   2 IS TRUE BRANCH
                                                               ** DD-PATH   3 IS FALSE BRANCH
                       . . . .
  29 ( 3)              IF BYTE ($ I + 1 $) ( ARG ) EQ O(20) $
                                                               ** DD-PATH   4 IS TRUE BRANCH
                                                               ** DD-PATH   5 IS FALSE BRANCH
                         . . . .
  31 ( 4)                FOR G = I , 1 , 11 $
  32 ( 5)                  BYTE ($ G $) ( ARG ) = O(00) $
                                                               ** DD-PATH   6 IS LOOP ON FOR AGAIN
                                                               ** DD-PATH   7 IS ESCAPE FOR LOOP
                     . . . .
  37 ( 2)            END
                                                               ** DD-PATH   8 IS LOOP ON FOR AGAIN
                                                               ** DD-PATH   9 IS ESCAPE FOR LOOP
                   . . . .
  39 ( 1)          IF PLACE EQ 12 $
                                                               ** DD-PATH  10 IS TRUE BRANCH
                                                               ** DD-PATH  11 IS FALSE BRANCH
                     . . . .
  41 ( 2)            FOR K = 0 , 1 , 11 $
  42 ( 3)              BEGIN
  43 ( 3)              IF BYTE ($ K $) ( ARG ) EQ O(20) $
                                                               ** DD-PATH  12 IS TRUE BRANCH
                                                               ** DD-PATH  13 IS FALSE BRANCH
                         . . . .
  45 ( 4)                IF PP $
                                                               ** DD-PATH  14 IS TRUE BRANCH
                                                               ** DD-PATH  15 IS FALSE BRANCH
                           . . . .
  48 ( 5)                  FOR J = K , 1 , 11 $
  49 ( 6)                    BYTE ($ J $) ( ARG ) = O(00) $
                                                               ** DD-PATH  16 IS LOOP ON FOR AGAIN
                                                               ** DD-PATH  17 IS ESCAPE FOR LOOP
                       . . . .
  55 ( 3)          K1.     END
                                                               ** DD-PATH  18 IS LOOP ON FOR AGAIN
                                                               ** DD-PATH  19 IS ESCAPE FOR LOOP
                   . . . .
  59 ( 1)          S1.   FOR I = 0 , 1 , 10 $
  60 ( 2)            BEGIN
                     . . . .
  62 ( 2)            IF BYTE ($ I $) ( ARG ) EQ O(00) $
                                                               ** DD-PATH  20 IS TRUE BRANCH
                                                               ** DD-PATH  21 IS FALSE BRANCH
                     . . . .
  64 ( 2)            IF BYTE ($ I $) ( ARG ) NQ O(52) $
                                                               ** DD-PATH  22 IS TRUE BRANCH
                                                               ** DD-PATH  23 IS FALSE BRANCH
                     . . . .
  69 ( 2)          S2.   END
                                                               ** DD-PATH  24 IS LOOP ON FOR AGAIN
                                                               ** DD-PATH  25 IS ESCAPE FOR LOOP
                   . . . .
  71 ( 1)          S3.   IF PLACE EQ 0 $
                                                               ** DD-PATH  26 IS TRUE BRANCH
                                                               ** DD-PATH  27 IS FALSE BRANCH
                     . . . .
  77 ( 1)            IF PLACE EQ 11 $
                                                               ** DD-PATH  28 IS TRUE BRANCH
                                                               ** DD-PATH  29 IS FALSE BRANCH
                     . . . .
  83 ( 1)            IF PLACE EQ 0 $
                                                               ** DD-PATH  30 IS TRUE BRANCH
                                                               ** DD-PATH  31 IS FALSE BRANCH
                     . . . .
  96 ( 1)          S9.   IF NOT KK $
                                                               ** DD-PATH  32 IS TRUE BRANCH
                                                               ** DD-PATH  33 IS FALSE BRANCH
-------------------------------------------------------------------------------------------
```

Figure 4.5. DD-path Definition Listing for DECMAL

```
PICTURE WITH ALL DD-PATHS...

MODULE <DECMAL  >, JAVSTEXT <BLSTIC  >, PARENT MODULE <BLSTIC  >

(B = BEGIN, E = END, S = SELF-LOOP)          STMT TYPE   STMT NO.          DD-PATH NUMBERS...
                                             <PROC    1> B                  1
                                                         +
                                           E <IF      26> EBB               2  3
                                             +              ++
                                             + <IF      29> BE+B              4  5
                                             +              + ++
                                            S+ <ASMT    32> EB++              6  7
                                             +              +++
                                             B <END     37> B+E+              9  8
                                                            ++ +
                                               <IF      39> E+B+B             10 11
                                                            ++++
                                           E <IF      43> B+E++B            12 13
                                             +              ++ +++
                                             + <IF      45> E+B+++B           14 15
                                             +              +++++
                                            S+ <ASMT    49> B+E++++           16 17
                                             +              ++ ++++
                                             B <END     55> ++B++EE           19 18
                                                            +++++
                                           E <IF      62> EE+EEBB           20 21
                                             +                 +  ++
                                             + <IF      64> BB+  +E           22 23
                                             +                 +++  +
                                             B <END     69> E++B E            25 24
                                                            +++
                                               <IF      71> BE+EB             26 27
                                                            + + +
                                               <IF      77> +B+BE             28 29
                                                            ++++
                                               <IF      83> E++EBB            30 31
                                                            ++ ++
                                               <IF      94> BE+BEE            32 33
                                                            + ++
                                               <END     97> E EE
```

Figure 4.6. Control Flow Picture for DECMAL

```
ITERATIVE REACHING SET
   OF DD-PATH   6 FROM DD-PATH   1

MODULE <DECMAL  >, JAVSTEXT <BLSTIC  >, PARENT MODULE <BLSTIC  >

   1 ( 0)            PROC DECMAL ( INPUT1 ) $                                  BEGINNING DDP
                                                                  ** DD-PATH   1 IS PROCEDURE ENTRY
                  . . .
  15 ( 1)               BEGIN
                  . . .
  18 ( 1)               ARG = INPUT1 $
  19 ( 1)               KK = 0 $
  20 ( 1)               PP = 0 $
  21 ( 1)               INTGRH = 12H(              ) $
  22 ( 1)               FRCTNH = 12H(              ) $
  23 ( 1)               FOR I = 0 , 1 , 11 $
  24 ( 2)                 BEGIN
                  . . .
  25 ( 2)                   IF BYTE ($ I $) ( ARG ) EQ D(33) $
                                                                  ** DD-PATH   2 IS TRUE BRANCH
                                                                  ** DD-PATH   3 IS FALSE BRANCH
  27 ( 3)                     BEGIN
  28 ( 3)                     PLACE = I $
  29 ( 3)                     IF BYTE ($ I + 1 $) ( ARG ) EQ D(20) $
                                                                    *ESSENTIAL PREDICATE*
                                                                  ** DD-PATH   4 IS TRUE BRANCH
  30 ( 4)                       BEGIN
  31 ( 4)                       FOR G = I , 1 , 11 $
  32 ( 5)                         BYTE ($ G $) ( ARG ) = D(00) $
                                                                    *ESSENTIAL PREDICATE*  ENDING DDP
                                                                  ** DD-PATH   6 IS LOOP ON FOR AGAIN
                  . . .
  35 ( 2)               PLACE = 12 $
  37 ( 2)               END
                                                                    *ESSENTIAL PREDICATE*
                                                                  ** DD-PATH   8 IS LOOP ON FOR AGAIN
```

Figure 4.7. Iterative Reaching Set for DECMAL

GENERAL CROSS REFERENCE LISTING

FOR WHOLE LIBRARY

SYMBOL	MODULE	USED/SET/DEFINITION (* INDICATES SET, D INDICATES DEFINITION)
A1	BLSTIC	40* 45 49 51 56
A9	BLSTIC	54* 77
AGAIN	BLSTIC	7D 86 87
ALOG	BLSTIC	49 49
ARG	DECMAL	4D 18* 26 29 32 43 49 56 62 64 67 76
ATAN	BLSTIC	53
B1	BLSTIC	41* 45
BHOL	CONVRT	3D 11 13* 18 25 29 39
C1	BLSTIC	27D 41 42 43 66
C2	BLSTIC	43* 49 53
C3	BLSTIC	42* 49
CON	FLTOUT	6D 17* 44* 55* 55 63* 63 66 68 70 76 78
CONVRT	CONVRT	1 4D 11 34* 40* 49
	DECMAL	56 79 88 89
COS	BLSTIC	52
COUNT1	DECMAL	12D
COUNT2	DECMAL	13D
CT	BLSTIC	53* 55 72 74* 74 75
CTH	BLSTIC	34D
D1	BLSTIC	44* 49 51
DA	BLSTIC	51* 52 52 53
DEC	FLTOUT	4D 50* 53 56* 56 58* 61 64* 64
DECMAL	DECMAL	1 2D 56* 79* 93* 96* 96
	INFO	7
E1	BLSTIC	31D 46 47 48 48
F1	BLSTIC	35* 39 40 54
FLTOUT	BLSTIC	57 60 62 64 66 68 70 72 75 77
	FLTOUT	1
FRCTN	DECMAL	9D 89* 91
FRCTNF	DECMAL	11D 91* 92* 92 93
FRCTNH	DECMAL	7D 22* 82 89
G1	BLSTIC	28D 45 49 53
G2	BLSTIC	29D 49
G3	BLSTIC	30D 41 42 43
H1	BLSTIC	32D 40 41 41 42 43 43 44 47
HOLLY	BLSTIC	5D 57* 60* 62* 64* 66* 68* 70* 72* 75* 77*
	LOAD	4
INFO	BLSTIC	35 36 37 38
	INFO	1
INPUT1	BLSTIC	6D
	DECMAL	1 3D 18
	FLTOUT	1 14D 18 101
	INFO	6 7
INTGR	DECMAL	8D 85* 88* 90
INTGRF	DECMAL	10D 90* 93
INTGRH	DECMAL	6D 21* 73* 76 79 88
I020	BLSTIC	82
I021	BLSTIC	84
I022	BLSTIC	86
I025	BLSTIC	85
I030	BLSTIC	24
I031	BLSTIC	26
I04000	BLSTIC	80

Figure 4.8. Library Cross Reference

```
MODULE INVOCATION SPACE...
MODULE <DECMAL  >, JAVSTEXT <BLSTIC  >, PARENT MODULE <BLSTIC  >
-------------------------------------------------------------------------------------------------------------------------------
  1             PROC DECMAL ( INPUT1 ) R                                                          (  1)
-------------------------------------------------------------------------------------------------------------------------------
INVOCATIONS FROM WITHIN THIS MODULE
-----------------------------------
.
MODULE CONVRT
                    STMT =   56         CONVRT ( ARG )
                    STMT =   79         CONVRT ( INTGRH )
                    STMT =   88         CONVRT ( INTGRH )
                    STMT =   89         CONVRT ( FRCTNH )
-------------------------------------------------------------------------------------------------------------------------------
INVOCATIONS TO THIS MODULE FROM WITHIN LIBRARY
----------------------------------------------

MODULE INFO
                    STMT =    7         DECMAL ( INPUT1 )
-------------------------------------------------------------------------------------------------------------------------------
```

Figure 4.9. Module Invocation Space for DECMAL

```
MODULE INVOCATION BANDS
MODULE <DECMAL  >, JAVSTEXT <BLSTIC  >, PARENT MODULE <BLSTIC  >
LEVEL -5        -4        -3        -2        -1         0         1         2         3         4         5
-------------------------------------------------------------------------------------------------------------------------------
                                                      DECMAL
                                        INFO                    CONVRT
                              BLSTIC
-------------------------------------------------------------------------------------------------------------------------------
```

Figure 4.10. Module Invocation Bands for DECMAL

JAVS REPORT FOR DD-PATHS NOT EXECUTED. 11 CASE(S) 10/02/75

MODULE NAME	JAVS TEXT NAME	TEST IDENTIFICATION		PATHS NOT HIT	LIST OF DECISION-TO-DECISION PATHS NOT EXECUTED
BLSTIC	BLSTIC	TSTCASE1 1	10/02/75	2	12 13
		TSTCASE2 2	10/02/75	6	1 2 3 4 5 13
		TSTCASE2 3	10/02/75	6	1 2 3 4 5 13
		TSTCASE2 4	10/02/75	6	1 2 3 4 5 13
		TSTCASE2 5	10/02/75	6	1 2 3 4 5 13
		TSTCASE2 6	10/02/75	6	1 2 3 4 5 13
		TSTCASE2 7	10/02/75	6	1 2 3 4 5 13
		TSTCASE2 8	10/02/75	6	1 2 3 4 5 13
		TSTCASE2 9	10/02/75	6	1 2 3 4 5 13
		TSTCASE2 10	10/02/75	6	1 2 3 4 5 13
		TSTCASE2 11	10/02/75	5	1 2 3 4 5
CONVRT	BLSTIC	TSTCASE1 1	10/02/75	4	2 6 8 9
		TSTCASE2 2	10/02/75	3	6 8 9
		TSTCASE2 3	10/02/75	1	2
		TSTCASE2 4	10/02/75	4	2 6 8 9
		TSTCASE2 5	10/02/75	1	2
		TSTCASE2 6	10/02/75	1	2
		TSTCASE2 7	10/02/75	4	2 6 8 9
		TSTCASE2 8	10/02/75	4	2 6 8 9
		TSTCASE2 9	10/02/75	4	2 6 8 9
		TSTCASE2 10	10/02/75	1	2
		TSTCASE2 11	10/02/75	4	2 6 8 9
DECMAL	BLSTIC	TSTCASE1 1	10/02/75	13	2 4 5 6 7 11 15 19 23 26 28 30 33
		TSTCASE2 2	10/02/75	13	2 4 5 6 7 11 15 19 23 26 28 30 33
		TSTCASE2 3	10/02/75	11	4 6 7 11 15 19 23 26 28 30 33
		TSTCASE2 4	10/02/75	11	2 4 5 6 7 11 15 19 26 28 30
		TSTCASE2 5	10/02/75	8	4 6 7 11 15 19 23 33
		TSTCASE2 6	10/02/75	11	4 6 7 11 15 19 23 26 28 30 33
		TSTCASE2 7	10/02/75	9	5 11 15 19 23 26 28 30 33
		TSTCASE2 8	10/02/75	11	2 4 5 6 7 11 23 26 28 30 33
		TSTCASE2 9	10/02/75	12	2 4 5 6 7 11 15 23 26 28 30 33
		TSTCASE2 10	10/02/75	7	11 15 23 26 28 30 33
		TSTCASE2 11	10/02/75	11	2 4 5 6 7 11 23 26 28 30 33
		TOTAL NOT HIT		1	11
PAYOUT	BLSTIC	TSTCASE1 1	10/02/75	11	7 9 10 12 13 16 19 21 22 24 30
		TSTCASE2 2	10/02/75	11	7 9 10 12 13 16 19 21 22 24 30
		TSTCASE2 3	10/02/75	11	7 9 10 12 13 16 19 21 22 24 30
		TSTCASE2 4	10/02/75	9	9 10 12 13 16 19 21 22 24
		TSTCASE2 5	10/02/75	11	7 9 10 12 13 16 19 21 22 24 30
		TSTCASE2 6	10/02/75	11	7 9 10 12 13 16 19 21 22 24 30
		TSTCASE2 7	10/02/75	11	7 9 10 12 13 16 19 21 22 24 30
		TSTCASE2 8	10/02/75	9	7 9 10 12 13 16 21 22 30
		TSTCASE2 9	10/02/75	9	7 9 10 12 13 16 21 22 30
		TSTCASE2 10	10/02/75	11	7 9 10 12 13 16 19 21 22 24 30
		TSTCASE2 11	10/02/75	11	7 9 10 12 13 16 19 21 22 24 30
		TOTAL NOT HIT		7	9 10 12 13 16 21 22

Figure 4.11. Multiple Test Coverage Report

```
MODULE DD-PATH COVERAGE LISTING

MODULE <DECMAL  >, JAVSTEXT <BLSTIC  >, PARENT MODULE <BLSTIC  >

NO.  LVL          STATEMENT                                  DD-PATHS GENERATED                                  COVERAGE
 1 ( 0)           PROC DECMAL ( INPUT1 ) $
                  . . . .
                  . . . .
23 ( 1)           FOR I = 0 , 1 , 11 $
24 ( 2)             BEGIN
                    . . . .
                    . . . .
26 ( 2)             IF BYTE ($ I $) ( ARG ) EQ O(33) $
                                                      ** DD-PATH   2 IS TRUE BRANCH                              8
                                                      ** DD-PATH   3 IS FALSE BRANCH                           470
                      . . . .
                      . . . .
29 ( 3)               IF BYTE ($ I + 1 $) ( ARG ) EQ O(20) $
                                                      ** DD-PATH   4 IS TRUE BRANCH                              2
                                                      ** DD-PATH   5 IS FALSE BRANCH                             6
                        . . . .
                        . . . .
31 ( 4)                 FOR G = I , 1 , 11 $
32 ( 5)                   BYTE ($ G $) ( ARG ) = O(00) $
                                                      ** DD-PATH   6 IS LOOP ON FOR AGAIN                       10
                                                      ** DD-PATH   7 IS ESCAPE FOR LOOP                          2
                    . . . .
                    . . . .
37 ( 2)             END
                                                      ** DD-PATH   8 IS LOOP ON FOR AGAIN                      434
                                                      ** DD-PATH   9 IS ESCAPE FOR LOOP                         36
                  . . . .
                  . . . .
39 ( 1)           IF PLACE EQ 12 $
                                                      ** DD-PATH  10 IS TRUE BRANCH                             36
                                                      ** DD-PATH  11 IS FALSE BRANCH                            0 *
                  . . . .
                  . . . .
41 ( 2)           FOR K = 0 , 1 , 11 $
42 ( 3)             BEGIN
43 ( 3)               IF BYTE ($ K $) ( ARG ) EQ O(20) $
                                                      ** DD-PATH  12 IS TRUE BRANCH                             46
                                                      ** DD-PATH  13 IS FALSE BRANCH                           199
                      . . . .
                      . . . .
45 ( 4)               IF PP $
                                                      ** DD-PATH  14 IS TRUE BRANCH                             32
                                                      ** DD-PATH  15 IS FALSE BRANCH                            14
                        . . . .
                        . . . .
48 ( 5)                 FOR J = K , 1 , 11 $
49 ( 6)                   BYTE ($ J $) ( ARG ) = O(00) $
                                                      ** DD-PATH  16 IS LOOP ON FOR AGAIN                      187
                                                      ** DD-PATH  17 IS ESCAPE FOR LOOP                         32
                    . . . .
                    . . . .
55 ( 3)             B1:
                    END
                                                      ** DD-PATH  18 IS LOOP ON FOR AGAIN                      209
                                                      ** DD-PATH  19 IS ESCAPE FOR LOOP                          4
```

Figure 4.12. DD-path Coverage for DECMAL

```
                    . . . .
59 ( 1)         S1.
                FOR I = 0 , 1 , 10 $
60 ( 2)           BEGIN
                  . . . .
62 ( 2)           IF BYTE ($ I $) ( ARG ) EQ O(00) $
                                                    ** DD-PATH  20 IS TRUE BRANCH          330
                                                    ** DD-PATH  21 IS FALSE BRANCH         100
                  . . . .
64 ( 2)           IF BYTE ($ I $) ( ARG ) NQ O(52) $
                                                    ** DD-PATH  22 IS TRUE BRANCH           99
                                                    ** DD-PATH  23 IS FALSE BRANCH          1
                  . . . .
69 ( 2)         S2.
                END
                                                    ** DD-PATH  24 IS LOOP ON FOR AGAIN    390
                                                    ** DD-PATH  25 IS ESCAPE FOR LOOP      39
                . . . .
71 ( 1)         S3.
                IF PLACE EQ 0 $
                                                    ** DD-PATH  26 IS TRUE BRANCH           1
                                                    ** DD-PATH  27 IS FALSE BRANCH         39
                . . . .
77 ( 1)         IF PLACE EQ 11 $
                                                    ** DD-PATH  28 IS TRUE BRANCH           1
                                                    ** DD-PATH  29 IS FALSE BRANCH         38
                . . . .
83 ( 1)         IF PLACE EQ 0 $
                                                    ** DD-PATH  30 IS TRUE BRANCH           1
                                                    ** DD-PATH  31 IS FALSE BRANCH         38
                . . . .
95 ( 1)         S9.
                IF NOT KK $
                                                    ** DD-PATH  32 IS TRUE BRANCH          39
                                                    ** DD-PATH  33 IS FALSE BRANCH         1

TOTAL DDPATHS              33
DDPATHS EXECUTED           32
PERCENT EXECUTED           96
```

Figure 4.12. (Cont.)

```
MODULE DD-PATH COVERAGE LISTING

MODULE <FLTOUT  >, JAVSTEXT <BLSTIC  >, PARENT MODULE <BLSTIC  >

 NO.  LVL          STATEMENT                          DD-PATHS GENERATED                          COVERAGE
---------------------------------------------------------------------------------------------------------
   1 ( 0)          PROC FLTOUT ( INPUT1 = SEQNM ) $
                   . . . .
                   . . . .
                                                      ** DD-PATH   1 IS PROCEDURE ENTRY            110
                   . . . .
                   . . . .
  22 ( 1)          FOR I = 3 , 1 , 15 $
  23 ( 2)            BYTE ($ I $) ( SEQNM ) = O(00) $
                                                      ** DD-PATH   2 IS LOOP ON FOR AGAIN         1320
                                                      ** DD-PATH   3 IS ESCAPE FOR LOOP            110
                   . . . .
                   . . . .
  25 ( 1)          IF OUTPT EQ 0 $
                                                      ** DD-PATH   4 IS TRUE BRANCH                 25
                                                      ** DD-PATH   5 IS FALSE BRANCH               85
                   . . . .
                   . . . .
  31 ( 1)          IF OUTPT GR 0 $
                                                      ** DD-PATH   6 IS TRUE BRANCH                 84
                                                      ** DD-PATH   7 IS FALSE BRANCH                1
                   . . . .
                   . . . .
  40 ( 1)      J1:
                   IF YY GQ 1.E-001 $
                                                      ** DD-PATH   8 IS TRUE BRANCH                 85
                                                      ** DD-PATH   9 IS FALSE BRANCH                0 *
                   . . . .
                   . . . .
  43 ( 2)            IF YY LS 1.E+000 $
                                                      ** DD-PATH  10 IS TRUE BRANCH                 0 *
                                                      ** DD-PATH  11 IS FALSE BRANCH               85
                   . . . .
                   . . . .
  52 ( 1)          FOR I = 0 , 1 $
  53 ( 2)            BEGIN
  54 ( 2)            IF YY GQ DEC $
                                                      ** DD-PATH  12 IS TRUE BRANCH                 0 *
                                                      ** DD-PATH  13 IS FALSE BRANCH                0 *
                   . . . .
                   . . . .
  60 ( 1)          FOR I = 0 , 1 $
  61 ( 2)            BEGIN
  62 ( 2)            IF YY LS DEC $
                                                      ** DD-PATH  14 IS TRUE BRANCH                85
                                                      ** DD-PATH  15 IS FALSE BRANCH              261
                   . . . .
                   . . . .
  69 ( 1)          IF CON - TEM EQ 0 $
                                                      ** DD-PATH  16 IS TRUE BRANCH                 0 *
                                                      ** DD-PATH  17 IS FALSE BRANCH               85
                   . . . .
                   . . . .
  74 ( 1)          IF PRE ($ 15 $) - TEM EQ 0 $
                                                      ** DD-PATH  18 IS TRUE BRANCH                83
                                                      ** DD-PATH  19 IS FALSE BRANCH               2
                   . . . .
```

Figure 4.13. DD-path Coverage for FLTOUT

```
77 ( 1)          J18.
                   IF CON / 100 EQ 0 $
                                                  ** DD-PATH  20 IS TRUE BRANCH          85
                                                  ** DD-PATH  21 IS FALSE BRANCH          0 *
                   . . . .
81 ( 1)          J20.
                   IF YY LS 1.E-001 $
                                                  ** DD-PATH  22 IS TRUE BRANCH          0 *
                                                  ** DD-PATH  23 IS FALSE BRANCH         85
                   . . . .
86 ( 1)            IF YY GQ 1000000000.E+000 $
                                                  ** DD-PATH  24 IS TRUE BRANCH          2
                                                  ** DD-PATH  25 IS FALSE BRANCH         83
                   . . . .
92 ( 1)          J22.
                   FOR I = 10 , - 1 , 3 $
93 ( 2)              BEGIN
                     . . . .
96 ( 2)              IF PRE ($ I + 1 $) - TEM EQ 0 $
                                                  ** DD-PATH  26 IS TRUE BRANCH        476
                                                  ** DD-PATH  27 IS FALSE BRANCH       204
                     . . . .
101 ( 2)         J28.
                   END
                                                  ** DD-PATH  28 IS LOOP ON FOR AGAIN  595
                                                  ** DD-PATH  29 IS ESCAPE FOR LOOP     85
102 ( 1)         J28.
                   IF INPUT1 LS 0 $
                                                  ** DD-PATH  30 IS TRUE BRANCH          1
                                                  ** DD-PATH  31 IS FALSE BRANCH       109

TOTAL DDPATHS          31
DDPATHS EXECUTED       24
PERCENT EXECUTED       77
```

Figure 4.13. (Cont.)

4.1.2 JAVS Self-Tests

JAVS self-tests demonstrated the capabilities of JAVS as a testing tool on a large (over 30,000 statements), complex program which had been previously subjected to a comprehensive set of test data using traditional test techniques. The test teams included personnel who designed and implemented a large portion of the JAVS software. They were familiar with both its functional specifications and its software organization. In addition, they had designed the test data during the software implementation activity.

JAVS software is a highly modular collection of hundreds of individual procedures, organized functionally into components. Several components are used together in each standard processing step. Table 4.1 shows the relation between the twelve components and the six processing steps. Each component's

TABLE 4.1

SUMMARY OF JAVS COMPONENTS USED IN STANDARD PROCESSING STEPS

	JAVS Component	Nucleus	Source Text Processing: BASIC	Source Text Processing: STRUCTURAL	Source Text Processing: INSTRUMENT	Test Preparation and Analysis: ASSIST	Test Preparation and Analysis: DEPENDENCE	Test Preparation and Analysis: ANALYZER
JAVS-0	Command & Control	●	●	●	●	●	●	●
JAVS-1	Storage Manager	●	●	●	●	●	●	●
JAVS-2	Primary Module Analysis		●					
JAVS-3	Secondary Module Analysis			●				
JAVS-4	Structural Analysis			●				
JAVS-5	Instrumentation				●			
JAVS-6	Data Collection and Reduction							●
JAVS-7	Test Case Assistance					●		
JAVS-8	Segment Analysis					●		
JAVS-9	Program Modification Analysis						●	
JAVS-10	Data Base Services	●	●	●	●	●	●	●
JAVS-11	Support Subroutines	●	●	●	●	●	●	●

interfaces with the other components are specified in its COMPOOL as procedure declarations; all inter-component communication of data space is passed through formal parameter lists. Within a component, the communication of data space includes both formal parameter lists and component-wide data specified by the COMPOOL in labeled common blocks. Data communicated between JAVS standard processing steps is wholly contained within the data base library, which resides on auxiliary storage.

In conducting the self-test portion of the acceptance tests, the following guidelines were used:

1. Execute 85% of the executable JAVS source code, excluding calls to error routines, with a minimal number of test cases. Note that the coverage requirement was given in terms of statements rather than DD-paths.

2. Utilize collateral testing whenever possible to minimize the resources required to run tests and analyze the results.

Selecting Test Data. Extensive test data was available to the test team from the normal implementation tests. This test data included a fairly short dummy program, TESTALL, which contains examples of most legal JOVIAL constructs (i.e., symbol type, statement type, control structure type, and module type) analyzed by JAVS. TESTALL was designed specifically for the functional demonstration portion of the JAVS acceptance tests. Since a large portion of JAVS software is itself functionally related to the JOVIAL constructs, TESTALL was an obvious choice for the test data for initial test execution.

Testing Procedure. Testing all JAVS software at once was impractical because of its size (in the instrumented form produced by INSTRUMENT). Since JAVS is a collection of twelve well-defined components, several of which are executed during each processing step, each component was separately analyzed by BASIC and STRUCTURAL, then instrumented by INSTRUMENT. During Test Execution, the remaining uninstrumented components were loaded as externals. Thus each test executed all of JAVS, but only one or two components at a time were instrumented. In some cases, subcomponents were defined and instrumented separately. Test Execution took the form of retrieving the instrumented component, satisfying externals from the remaining uninstrumented JAVS components, and executing an appropriate processing step on TESTALL. For example, to test the JAVS-2 component, the processing step executed was BASIC, since BASIC uses components JAVS-0, JAVS-1, JAVS-2, JAVS-10, and JAVS-11. The ANALYZER module coverage reports dealt only with the modules in JAVS-2 which were instrumented.

Each component was analyzed in this manner. The well-defined structure of the JAVS software, coupled with a fairly exhaustive JOVIAL test case, provided better than the required 85% statement coverage, and a DD-path coverage of 70%, in two test runs (only one test run for all components, except certain subcomponents of JAVS-2). The detailed results of the JAVS self-tests are included in Ref. 7.

The high coverage with a single set of test data is attributed to several factors: the design of JAVS software as a set of functionally related components, the use of structured programming in implementing the software (except for JAVS-2), the availability of a well-designed set of test data, and the test team's intimate knowledge of the software and its usage.

Automated Documentation. In addition to satisfying the self-testing requirements, processing the JAVS components through the JAVS system provided comprehensive computer documentation of each module and component. Computer documentation consists of module listing, module DD-path listing, module control-flow picture, component (symbol) cross reference, and component module-dependence tables. The module-dependence tables were helpful in determining subcomponent grouping for JAVS-2. The cross-reference listings, along with the module statement coverage reports, were useful for determining the second test case that was needed for some of the JAVS-2 modules.

In addition, the module dependence summary identified several superfluous modules which were not called by other modules. This documentation is both comprehensive and accurate; it has continued to prove its value in the maintenance of JAVS software during the current contract.

Summary. The JAVS self-tests demonstrated the following:

- Applying JAVS to a large test object is feasible.
- Well-structured software, although large, lends itself well to AVS test techniques.
- Software implemented to reflect functional requirements achieves high coverage when the test data covers the functions.
- The high-quality documentation produced by JAVS is invaluable in both testing and maintaining software.
- The test team's knowledge of both the design of the software test object and how to use it, as well as how to use JAVS, contributes immeasurably to the success and efficiency of testing.

4.2 EVALUATION TESTS

The primary objective of the current contract was to implement a systematic software test program using JAVS to assist in the testing process.[6] The bulk of the effort was spent on testing the COMPOSE subsystem of SAC's Force Management Information System (FMIS), a generalized data management software system. The general characteristics of FMIS are (1) it is large, (2) it is organized into modular components, (3) it operates interactively through a command language, (4) it manipulates a complex data base, and (5) its code is not structured code.

Before testing began, a training course was conducted to acquaint the test team with JAVS and the testing methodology. Next, a formal test plan

was prepared which emphasized the evaluation of JAVS by using it to test and verify a large, operational software system.

COMPOSE was selected as the test object solely because it was a readily available, large, complex program written in JOVIAL J3. COMPOSE is only part of a very large software system. Although the source code for COMPOSE was available, the source code for other parts of the FMIS system was not provided; in particular, that for the COMPOOLs which specify interfaces and define data structures. This lack of complete information hampers the testing when the software utilizes global data for communication between subsystems, as FMIS does. Furthermore, nothing was known by the test team about the size, structure, or operational behavior of COMPOSE at the beginning of testing. In addition the test team had no actual experience in operating FMIS or in using its command language and the (supplied) data base.

Preliminary Testing Procedure. Because COMPOSE has more than the 250 modules permitted for a single JAVS library, it was divided into seven components (with minor overlap), ranging from 52 to 218 modules in size. Each component was treated as a separate library, passing first through BASIC and STRUCTURAL, then through ASSIST and DEPENDENCE for documentation reports. The resulting libraries contained a total of 705 invokable modules and 38,734 JOVIAL statements excluding comments.

The size and complexity of COMPOSE stressed the JAVS syntax analyzer, exposed some incompatibilities between the syntax analyzer and the JOCIT compiler, violated JAVS size limitations and language usage constraints, revealed JOCIT compiler malfunctions in compiling JAVS software, exceeded external symbol table size limitations for JOCIT compilations, and uncovered some errors in JAVS code. Concurrent with this effort to build libraries and document COMPOSE code, procedural techniques in using JAVS were standardized, processing improvements were made to JAVS software, and errors in JAVS were corrected.

Selecting Test Points. JAVS testing coverage analysis requires the user to identify specific statements which correspond to the beginning and end of a test. Proper selection of these test boundary points is based on knowledge which relates program behavior to software implementation. Lack of familiarity with the COMPOSE software and its behavior by the test team initially resulted in improper placement of the test points. For a program which is segmented into memory overlays, logical test points occur at link loading events. JAVS documentation of COMPOSE was used to locate the statements which caused link loading, and appropriate code was added at those points to capture these events on the test file. Since the test file normally contains a time-sequence record of program behavior (e.g., module invocations and returns), selection of appropriate test points was greatly simplified by analyzing how the link loading events were interspersed with recorded test events. In this respect, JAVS serves to acquaint the tester with program behavior.

Test Environment. FMIS normally operates interactively under the time-sharing facilities of the operating system. The user supplies a data base, enters a series of commands to FMIS, and receives responses from FMIS on an interactive terminal. For test purposes this type of interaction is not

always possible nor is it necessarily desirable. The overhead of code expansion through instrumentation, as well as added processing time, can be prohibitive under time-sharing constraints. Furthermore, the need to record and correlate all input data with test results and possibly to duplicate a test precisely (e.g., without transmission errors or wrong key strokes) demands a controlled test environment. For these reasons, COMPOSE was tested in batch mode.

However, the batch mode of operation presented additional problems. Due to lack of familiarity of the test team with preparing input commands for FMIS, coupled with the self-protection of the software against erroneous data, it was difficult to prepare data which resulted in completed tests. As a result, test data was first designed, executed with the normal interactive version of COMPOSE, then later entered into the test version in batch mode.

Test Procedure and Results. The software on each library which was tested processes a specific subset of FMIS commands. Test data was prepared and test coverage reports were examined to determine testing targets: modules with little or no coverage and key DD-paths which were not exercised. DD-path testing targets were determined according to several criteria: deep nesting level in the control structure, as far into the code sequence as possible, and (as frequently was the case in testing COMPOSE) identification of variables in the predicate of untested DD-paths which are known to be directly affected by the input data.

Generating new test cases to achieve better coverage can be very difficult when the test program is large and complex. When the input data undergoes substantial processing by the source code before it enters the software component being tested, the task is even harder. For COMPOSE, this was further complicated since many of the variables found to affect DD-path testing targets are global variables which can be modified outside of the test component. The lack of complete documentation (e.g., COMPOOL source code) adds to the problem. In testing COMPOSE under these conditions, the most effective testing assistance was provided by the combination of the JAVS test analysis reports which show the source code executed; the JAVS symbol cross-reference report showing library-wide references; and the SAC documentation manual describing the functions of the modules. With this procedure the initial tests with two components achieved 29% and 48% DD-path coverage. Modifying the input and retesting these two components improved the coverages to 41% and 54% respectively. By analyzing post-test reports together with the SAC documentation, many of the untested DD-paths were shown to handle error conditions, boundary conditions, improbable input combinations, or a more complex data base.

Many errors in the COMPOSE maintenance manual (mostly in intermodule dependencies) were identified by comparing the JAVS-processed documentation

of the actual software to that in the maintenance manual. The availability of high-quality, accurate, up-to-date documentation significantly improves the efficiency of both testing and maintenance.

Summary. COMPOSE testing emphasized the problems associated with bottom-up and top-down system testing strategies. Bottom-up testing generally requires special testing environments. Since most variables in COMPOSE are defined in the COMPOOLs, and the COMPOOL source text was not available, it was not possible to artifically create a correct test environment.

Top-down testing progresses from the modules which are highest in the invocation structure to those lowest in the invocation structure for the complete software system. A natural secondary progression is in order of module execution. Since none of the source code for the interface segments was available for either testing or documentation, the communication paths between much of the tested code and the input of data were blocked.

Although the retesting process proved difficult, the JAVS execution tracing and test coverage reports provided an accurate statement of program behavior. This output, together with the JAVS documentation reports, greatly accelerated the test team's learning about the software test object.

To evaluate JAVS's contribution to software maintenance, typical maintenance problems were posed: for example, the problem of modifying the effect of a particular command (e.g., PROOF). The SAC documentation was used to identify the module whose function it was to process the command, the JAVS intermodule dependence reports identified the modules directly affected (by invocation) by the proposed change, and the cross-reference reports on the use of variables identified the modules indirectly affected (by commonality of reference to variables). The tester, using the JAVS reports alone, quickly and easily identified the code requiring change and determined the extent of affected modules.

5 APPLICATION OF SYSTEMATIC TESTING METHODOLOGY

Without an AVS, testing has suffered from lack of formalized, systematic, enforceable techniques. Testing with an AVS not only offers an accurate means of determining exactly what portions of the software have been exercised, but also assists in the preparation of test data. These capabilities greatly improve the effectiveness of testing. Side benefits of an AVS are automated, high-quality, accurate documentation of computer programs; reports that are useful for code optimization; and dynamic testing capability with embedded assertion statements for expected software behavior.

Previous sections have presented an overview of the testing methodology, the capabilities of one AVS implementation (JAVS), and actual testing experience with JAVS. This section considers the general role of an AVS in applying the testing methodology and discusses practical techniques for particular situations.

5.1 ROLE OF THE AVS

The concepts implemented in JAVS address the basic problem of assembling systems of hardware and software to achieve some desired behavior. Software designed from specifications usually exhibits behavior not included in the specification. The specified behavior may be acceptable (i.e., what was desired) or unacceptable (i.e., not desired because of unforeseen consequences of the system's behaving as specified). The unspecified behavior may be acceptable (i.e., fortuitously providing a capability not included in the specification) or unacceptable.

Traditionally an attempt is made to map this total system behavior by implementing the specification, then testing to establish the correctness of the implementation, as well as to identify unacceptable behavior for subsequent correction.

The process of ultimately achieving acceptable behavior in a software system can be divided into two approaches: attempting to build software of inherently high quality, and attempting to identify failure by testing the product at various stages. Figure 5.1 shows a few of the approaches that are used and proposed for improving software quality and arranges them according to where they fall with respect to synthesis and analysis (the two overlap depending on how they are applied). The approaches are also arranged in order of increasing time (i.e., increasing collective experience in building software).

The bold arrows show the relationships of the JOVIAL Automated Verification System to predecessor approaches. Figure 5.1 also shows how the other approaches relate to each other and to the concept of an advanced automated verification system.

Figure 5.2 shows a diagrammatic relationship of a tester to the test objects in the unaided testing process. This process, while imperfect and highly dependent on the skill of the tester, nonetheless is the experience base on which "better" approaches should build.

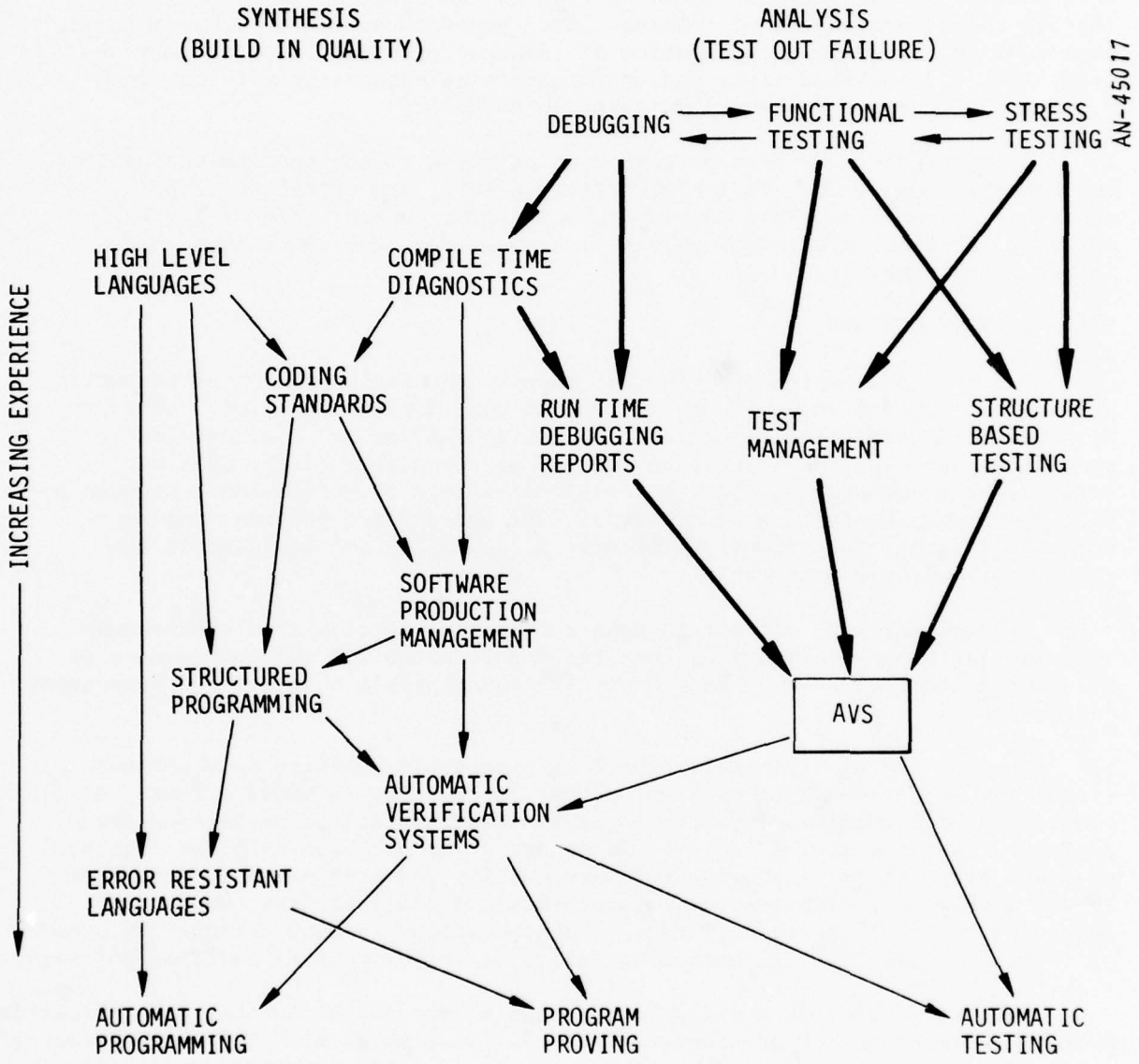


Figure 5.1. Approaches to Achieving Quality Software

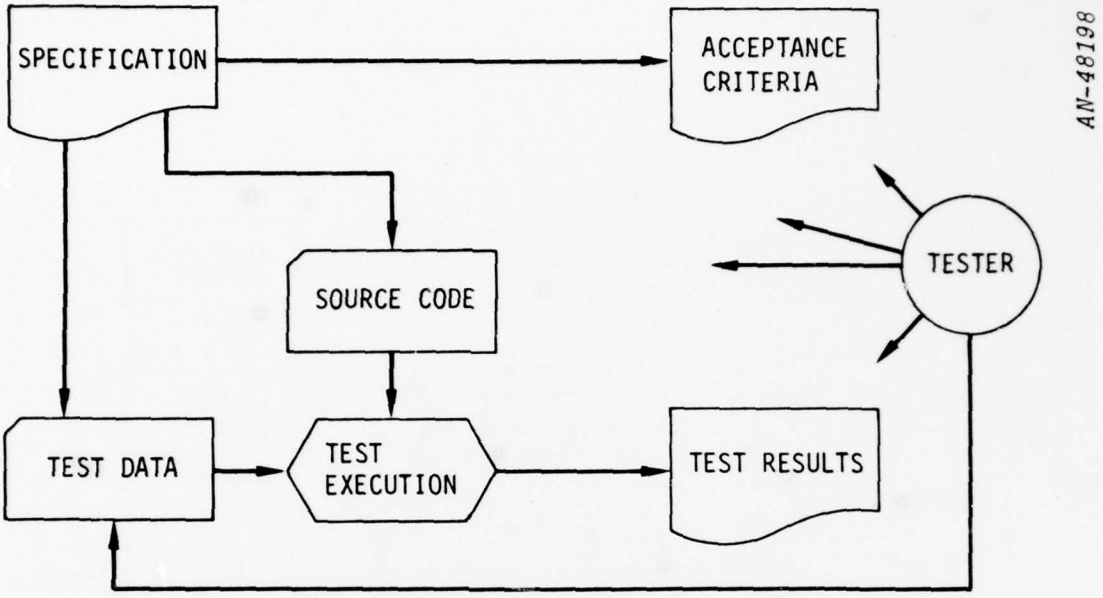


Figure 5.2. Unaided Software Analysis and Testing

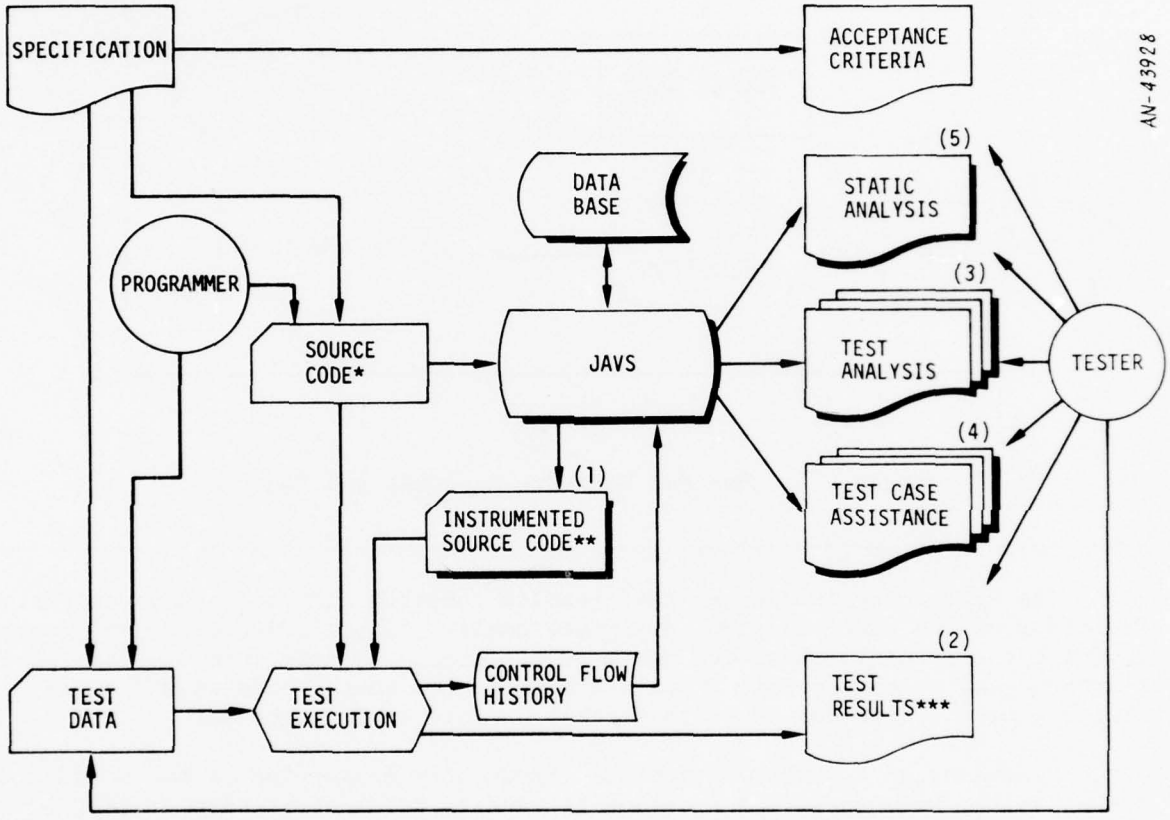
The tester, by looking at test results compared with acceptance criteria derived from the specification, evaluates quality of performance and the degree to which the specified behavior has been explored. By some more or less intuitive process, which probably entails examining the source code as well, the tester generates new test data to further explore system behavior.

Depending on a number of factors (frequently exhaustion of the test budget is the sole factor) a decision is made to terminate testing. If resources permit and the tester is clever, much of the total system behavior will be exposed, although in general it will not be known how much. In particular there may be little or no attempt to explore unspecified behavior.

Figure 5.3 illustrates the role of JAVS in software analysis and testing. The data base stores the source text being tested and the tables used in the analysis and report generation.

JAVS augments the testing process in the following ways:

- By automatically inserting code to collect data on control flow, a record of the portions of code that have been exercised is captured during testing. (See (1) in Fig. 5.3).



*INCLUDES COMPUTATION DIRECTIVES
**CONTAINS EXPANSION OF DIRECTIVES AS WELL AS STRUCTURAL INSTRUMENTATION PROBES
***NORMAL PROGRAM OUTPUT IN ADDITION TO RESULTS OF EXECUTION OF DIRECTIVES

Figure 5.3. Software Analysis and Testing Augmented by JAVS

- The status of selected variables and conditions is produced from expansion of the JAVS computation directives manually inserted by the user. (See (2) in Fig. 5.3.)

- Subsequent test analysis of the control flow data collected from the instrumented code exposes code that has not been exercised. (See (3) in Fig. 5.3.)

- Test case assistance reports aid in generating test cases to exercise untested code. (See (4) in Fig. 5.3.)

- Static analysis aids the test case generation process for both single-module and system-wide testing. (See (5) in Fig. 5.3.)

5.1.1 Relation Between Software Testing and Software Validation

Figure 5.4 describes the relationship between software testing and software validation. The ideal specification implementation would be achieved by automatic conversion of specifications to software, with the process validated by an alternate mapping of the software back to the specifications in a way that exposes not only the failure to properly implement specifications but also the introduction of spurious behavior in the software implementation.

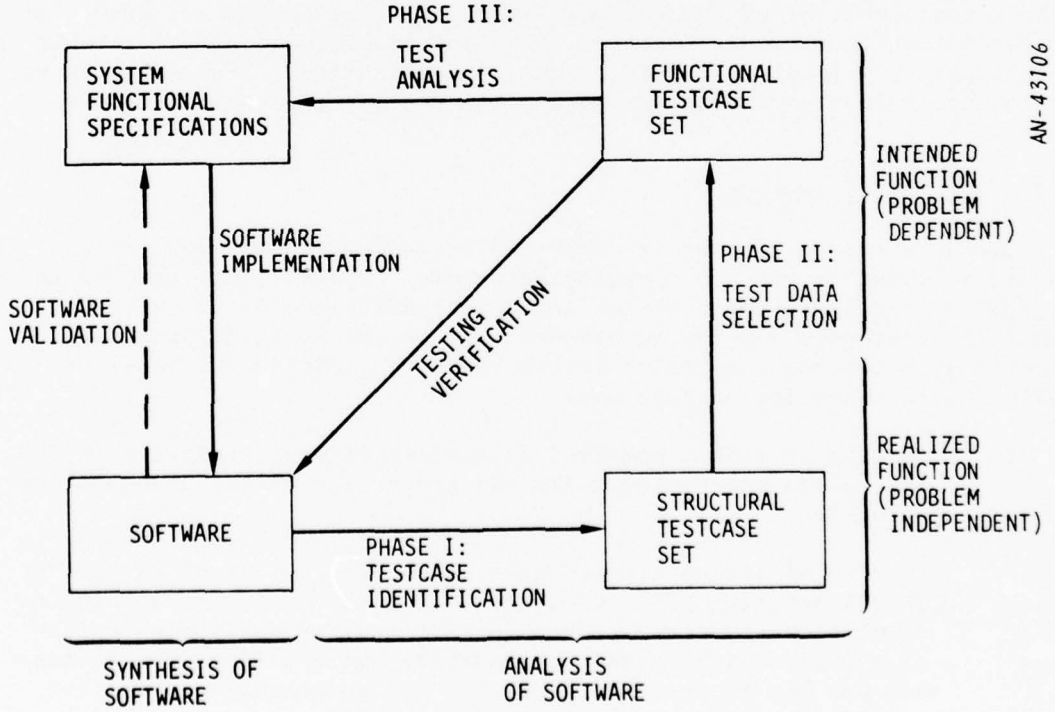


Figure 5.4. Software Testing and Validation

Neither of these processes is realizable today. Software implementation is largely a manual process. JAVS, however, provides automated tools to assist in the software validation process. This activity divides into three phases:

- Phase I: Structural Test Case Identification. The software is analyzed for specific kinds of information that yield a collection of structural tests.

- Phase II: Test Data Selection. Supplying specific input values for a structural test converts it to a functional test.

- Phase III: Test Analysis. The set of functional test cases for the software is then analyzed for its relationship to the System Specifications.

JAVS performs analysis functions needed in Phases I and II of the process.

The bridge between functional test case sets (the outcome of Phase II) and the behavior these functional tests evoke from the software system is called testing verification, shown in Fig. 5.4 as a diagonal line. Testing verification binds the problem-dependent and problem-independent aspects of the process together by providing the means to observe the actual behavior of a functional test case set derived purely from structural analysis. In addition, these executions of the software provide the measures of testing coverage that are necessary for subsequent application of the testing methodology. If the actual behavior of the software system differs in any way from what is expected during testing verification, then one has located either a fault in the software or a fault in the functional specification. Thus, testing verification provides a useful approximation to comprehensive test analysis (Phase III).

5.1.2 Overview of Testing

Software system testing is organized according to two hierarchies: the invocation structure and the decision structure. System-level testing is performed primarily in terms of the invocation structure of the software system; single-module testing is based wholly on the decision structure. Systematically testing a software system requires judiciously chosen use of the two distinct testing disciplines:

1. Testing of single modules, regardless of where they lie in the invocation structure, using the properties of the module's decision structure.

2. Testing of components and subsystems, regardless of where they lie in the decision structure, based on their positions within the invocation structure. This form of testing deals with collections of modules which invoke one another, but considers the actions of all but the topmost module within any component as primitive activities, to be tested according to (1).

The selection between (1) and (2) is based, at least in part, on measuring the level of testing coverage achieved as testing proceeds.

In either case, the software must first be prepared for testing (e.g., analyzed for its structural properties and instrumented for testing). This is done by processing the software through BASIC, STRUCTURAL, and INSTRUMENTATION (see Sec. 3). The resulting data base contains all information necessary for subsequent testing activities.

5.2 PRACTICING THE METHODOLOGY

How testing is to be accomplished is determined by answering very specific questions:

- What is the software test object? (Sec. 5.2.1)
- What are the available resources (e.g., hardware, support software, personnel, time, test tools)? (Sec. 5.2.2)
- What are the test goals? (Sec. 5.2.3)
- What procedure will accomplish the goals? (Sec. 5.2.4)

There is no single, general procedure which applies to all testing. Each particular testing activity is unique and should be analyzed to determine a suitable procedure. This suggests that the testing process itself consists of three distinct phases: (1) identifying the elements of the test activity, (2) preparing a test plan, and (3) carrying out the planned tests. The remainder of this section addresses the problem of practical application of the testing methodology.

5.2.1 Software Test Object

The software to be tested is called the software test object. The characteristics of the software important to AVS-supported testing are listed in Table 5.1. JAVS specifications and processing pertinent to software characteristics are also shown.

Source Language. The AVS deals with the software test object in source-language form. This is the form most suitable for AVS-supported testing because it is the form used to create and to modify the software. The AVS assumes that the source text is free from syntactical errors (i.e., it can be compiled without syntax-related errors).

Size. An AVS supports testing of both small and large software test objects, consisting of a single module or perhaps hundreds of modules, and totaling from a few statements to tens of thousands. The test object may consist of one or more compilation units. On a specific computer, an AVS will be limited by the size of the direct-access memory and the size of auxiliary files.

TABLE 5.1

CHARACTERISTICS OF SOFTWARE TEST OBJECT

Software Characteristics	JAVS Specifications
1. Source Language	JOVIAL/J3 (may have imbedded assembly-language code)
2. Size	Small to large
• Number of modules	1 to 250 per library
• Number of statements	Unlimited
• Compilation units	Single or multiple
3. Organization	
• Complete/incomplete program	One or more START-TERMS
• Software partitioning	Linked or non-linked
• Design	Assertion statements, automated documentation, dynamic test identification, modular, structured for best results
4. Computer System	HIS 6180, CDC 6400, HIS 6080
5. Support Softweare	
• Operating system	WWMCCS for HIS 6080, GCOS for HIS 6180, GOLETA for CDC 6400
• Communications	
• Compiler/assembler	JOCIT JOVIAL
• Link loader	System loader
• Library	JAVS probe routines, sequential file I/O
6. Suitability for Testing	
• Design	Non-recursive, non-concurrent, non-time-dependent, direct correspondence to functional specifications, traceability of symbols to input, identifiable I/O
• Partitioning	Functionally similar components
• Data	Initial test case

Organization. The software test object may be a complete or an incomplete program; it may be organized into one or more subsystems, or may be a utility package. If the program is incomplete, test execution will require additional software, either operational software or special test software, which combines with the software test object to result in a complete program. Although the additional software need not be processed by the AVS in source-language form, its interfaces with the software test object must be clearly and unambigously defined in order to prepare test data. Some source languages (e.g., JOVIAL, Pascal) require formal interface definitions for both data access and module invocation. The source text for these definitions is essential for AVS-supported testing.

For a very large software system consisting of many hundreds of modules, it is wise to partition the software into test objects of tractable size. Normally very large software systems are designed as subsystems according to some functional criteria. If the subsystems themselves are each a collection of hierarchically structured related components or modules, this same partitioning may also be suitable for testing purposes. Miscellaneous collections of low-level utility modules which are invoked throughout the remainder of the system can be grouped together as a separate subsystem.

In partitioning the software test object, consideration should also be given to the resources required to test each partition. Once it is instrumented the software requires more main memory because of the code expansion due to the instrumentation, and more computer time because of the overhead of executing the probes. A further consideration is the execution-time behavior of the software test object. During test execution it is important to designate important events (e.g., file activity, link loading, major cycles) which separate the test execution into a sequence of individual tests. This permits the tester to extract more detail from the test results. Candidate events include the start of an initialization or termination process, the start or conclusion of a new test case, the change in mode of behavior (e.g., from normal mode to error mode), opening or closing of files, memory link loading, and invocation of other software not being tested.

To properly partition large software test objects, the tester must use documentation supplied with the software for guidance.* This documentation, which may have been manually prepared, more often than not does not accurately reflect the current version of the software. The documentation capabilities of the AVS offer automated assistance in verifying the accuracy of the software documentation. For example, the intermodule dependence reports give a concise picture of the interface of one set of modules to all referenced modules. Trial partitions of the software may be defined by the user, and verified with these reports from the AVS. If the software is too large to be processed by the AVS as a single unit, initial partitioning must be based on the documentation supplied.

*See Sec. 5.2.4 for partitioning criteria according to software structure.

Computer System. The computer that the software test object normally operates in may also be suitable for testing. The computer used during test execution, however, must have sufficient excess resources to accommodate the code expansion due to instrumentation and recording of test probe data. The AVS may, or may not, execute on the same computer as the software under test. If an intermediate test file is used to record the results of test execution, only the AVS data collection routines directly interact with the software test object during test execution. JAVS requires only the data collection routines to reside on the same computer as the test object.

Support Software. The software test object may make use of support software which is not normally considered to be part of the application. For example, it may execute under the control of an operating system, with or without the support of communications software (i.e., as a time sharing application). A compiler (or assembler) is essential to translate source text into object text. Furthermore, support by a link loader and access to required library routines are also necessary. For AVS-supported testing, the same support software is used.

Suitability for Testing. There are additional characteristics of the software which affect its suitability to AVS-supported testing. These are a result of assumptions made in the AVS implementation itself. For example, the current JAVS implementation assumes that the software test object contains no recursion, has no concurrent paths during execution, and is not time-critical. Each of these restrictions may be relaxed in other AVS implementations.

Some software design characteristics facilitate AVS-supported testing. Among these are:

- Highly modular, structured code
- Direct correspondence of implemented software to functional specifications
- Localization of code controlling important events in software behavior (e.g., new test case, file I/O, link loading)
- Identifiable module inputs and outputs
- Mnemonic symbol names
- Traceability of symbols to input symbols

AVS-supported testing is hampered if the software test object lacks these, or if it contains logically unreachable code, uses borrowed code, or bypasses normal module invocation and return protocol for control transfer.

Software may also be deliberately designed to take advantage of specific AVS capabilities such as the use of imbedded assertion statements for dynamic checking of expected behavior, automated documentation, or program performance with test point identification.

5.2.2 Test Resources

The resources available for testing must be identified before an approach to testing can be determined. These include the computer resources, data for executing the software, the members of the test team, the AVS capabilities, and the time within which testing must be completed.

Computer Resources. All computer resources (both hardware and support software) used by the software test object during normal execution should be identified. This information is usually contained in software documentation or can be extracted from sample execution runs. For example, for programs which execute under the control of an operating system, the hardware and software requirements may be extracted from reports produced by the system loader and from job control statements. If the program is overlayed (i.e., different parts of the program reside in main storage at different times), then the memory layout of each overlay link is also needed. This mapping of the test software onto the computer is referred to as the execution environment of the program.

Test Execution replicates normal execution in the sense that the program reads its own inputs and produces its own outputs. Additional output is captured from the instrumented program during Test Execution for later analysis by the AVS. Test Execution differs from normal execution in the following ways:

- Some or all of the program has been instrumented to capture execution behavior on a test file
- Execution output will include the results of any assertion statements embedded in the source code
- Test case boundaries are identified at particular test points in the software
- The instrumented code, although logically equivalent to the uninstrumented code with probes added, contains references to AVS probe routines which are added to the load sequence
- The probe test file is recorded

The major effects from these software perturbations are the increased memory requirements and execution time due to instrumentation.

In some instances the additional computer resources are sufficient to preclude testing in the normal execution environment. Other considerations such as accessibility and operational compatibility with the AVS execution environment may also indicate a change in execution environment for Test Execution. For example, the FMIS software (see Sec. 4) normally operates in a time-sharing mode under the GCOS operating system. The expanded core requirements of the instrumented code, together with the need for rapid analysis of test file data by JAVS, made it more practical to test FMIS as a multiple-activity batch job for Test Execution. In this mode the first activity uses JAVS to instrument the desired component, the next compiles the instrumented

code, the next loads the test and executes the (partially) instrumented FMIS, and the last uses JAVS to analyze the test results.

Operational procedures must be suitably modified to accommodate the necessary changes. For programs which execute under the control of an operating system, these modifications include altering the job control language to provide for compiling instrumented code and merging it with non-instrumented code, loading the AVS probe routines, and capturing the test file for analysis by the AVS.

Data Requirements. For Test Execution, the software is exercised with test data in the test environment. This data may be generated specifically for the AVS-supported test activity, or may be taken from previous execution of the software, or both. Existing test data as well as results from tests unsupported by an AVS can be invaluable to the test process, especially if the tests reflect functional requirements of the software. For example, FMIS uses two types of input data: user commands and a complex data base. In AVS-supported testing, the data found to have the most direct effect on FMIS program coverage was the user command data, with the data base of lesser importance. Testing was conducted with an existing data base, holding that input fixed and altering only the user command data.

Test Team. Testing requires a test team capable of preparing data, making computer runs with the software, and analyzing output produced during execution tests. In addition, AVS-supported testing requires that the test team know how (1) to use the AVS capabilities for analyzing the software, (2) to make suitable modifications for test execution, and (3) to analyze the combined test results from the software and AVS (i.e., normal output from the software together with AVS post-test analysis of coverage derived from software probes). It is important that the test team have more than superficial knowledge about the software test object. In particular, the team must have specification-level knowledge of functional behavior, at least some knowledge of functional behavior, at least some knowledge of program structure, and detailed knowledge of operational requirements.

AVS Capabilities. For effective use of the AVS, the test team must understand the purpose of each of the AVS processing capabilities and select those that are applicable at each stage of testing. The AVS can be used to accelerate testing. For example, if the test team's knowledge of the software's structure is deficient due to lack of detailed documentation, prior to actual testing the AVS documentation capability can derive detailed information about inter-module dependencies directly from the software. The additional information needed about each module to attach meaning to the invocation structure includes each module's purpose, its inputs and outputs, and the interpretation associated with data processed by the module.

5.2.3 Test Goals

The overall goal of testing is to improve the quality of software through testing and validation of test results. Detailed testing goals are directly related to the type of testing to be done: single-module testing or system-wide testing. The type of testing, in turn, may depend on the stage of software development and previous test history of the software test object. It is often the case that single-module testing is most appropriate during the code development phase or whenever a module has been changed or replaced during the maintenance phase. System-wide testing is applicable whenever collections of modules are tested (e.g., in software integration and maintenance phases or in top-down development).

Single-Module Testing. For single-module testing, the testing process for complete coverage has a single objective: to construct a usefully small set of test cases which, in aggregate, cause execution of each DD-path in the module at least once. Real programs may have DD-paths which cannot be exercised, no matter what input values are used. An alternative goal, then, is to exercise each DD-path that can be exercised, at least once, and for each DD-path that cannot be exercised, provide a detailed explanation of why it cannot. Programs which have been tested to this level will meet the following criteria:

- Each statement in the program will have been executed at least once.
- Each decision in the program will have been brought to each of its possible outcomes at least once, although not necessarily in every possible combination.

More stringent test goals include exercising each outcome of each decision in a single test, exercising all possible levels of iteration, and exercising all possible program flows (the last is in general not possible).

System-Wide Testing. For system-wide testing, the testing goal is a straightforward extension of the single-module testing goal: to construct a usefully small set of test cases which exercise as many DD-paths as possible, out of the aggregate set of DD-paths in all modules in the system. The coverage measure may be an overall percentage of DD-paths exercised, the percentage of DD-paths exercised in the least tested module, or the percentage of DD-paths exercised in the least tested subsystem of modules. A more stringent test goal would measure coverage in relation to a module's location on the invocation hierarchy. For example, any module which is referenced by more than one other module might have its coverage separately determined for each referencing module.

5.2.4 Testing Strategy

The previous subsections have focused on defining the software test object and collecting information about its structure, its operation, and the effects of using an AVS in testing. This subsection presents a general methodology for testing a software system and gives guidelines for the

effective use of an AVS. Detailed information about the use of JAVS is contained in Refs. 4 and 5. The major steps in testing with an AVS are:

1. Understand the functional requirements of the software system
2. Generalize the modes of behavior of the system
3. Define the system's hierarchical structure
4. Develop test plans keyed to the modes of behavior and the software structure
5. Execute functional tests
6. Develop structure-based tests for increased coverage

System Behavior. Information about the software's functional requirements (i.e., the expected response of the system to inputs) is usually contained in the system specification documentation and the program documentation. Complex programs may have more than a single mode of behavior. For example, a data base management system may have a primary, high-priority mode which handles user commands interactively, and a secondary, background mode which generates periodic reports on system usage. Other modes of system behavior may include error processing or operation under degraded conditions (e.g., with an incomplete or garbled data base). If there is more than one mode of expected behavior, each should be identified and named.

System Structure. Having defined the system's modes of behavior, an attempt should be made to relate each mode to a hierarchical structure of the program. There are several kinds of structure in a computer program: data structure is the organization of the data on which the program operates; computation structure describes the program's operations on the data; and control structure is the kind directly dealt with by the testing methodology. The other kinds of structure are tested only insofar as they interact with the control structure.

Large software systems are usually organized into subsystems, subsystems into components, and components into modules. This static organization describes the way the individual elements of the system depend on one another, without regard to the way program control flows up and down. The dynamic organization of a software system is the structure which results from considering the effects of all invocations between modules, components, and subsystems. It is usually called the invocation hierarchy of the software system. The testing methodology deals with the dynamic organization. Normally program documentation will identify the static organization of software modules. The dynamic organization can be derived from more detailed documentation and verified by the AVS documentation capabilities for showing intermodule dependencies.

For each named mode of behavior a software description should be developed which identifies the modules invoked, the specified input domain of each module, and the expected system performance. Existing software documentation may not contain sufficient information to document each behavior mode separately. It

may be necessary to execute some preliminary functional tests with the software instrumented at the module invocation level to determine which modules are invoked and under what conditions.

Test Plans. To minimize testing effort, test plans should be developed which are keyed to the named modes of behavior and to software structure. Each test plan should identify one or more functional tests for initial testing and the software structures to be tested. It is often the case that more than one substructure of the software system will be exercised with a given test case. This so-called collateral testing can greatly reduce the overall testing effort. Each test plan should contain the following information for each functional test:

- A name identifying the test
- What function is tested
- The primary code structure tested
- Collateral code structures tested
- A description of the resources required
- The expected performance
- Criteria for evaluating the test

All of these items are commonly called for in test plans for software acceptance tests.[7] Wherever appropriate, use should be made of existing functional tests and test plans. The major distinction between testing with and without an AVS is that with an AVS there is an orderly progression of testing from the initial tests through well defined steps to achieve the desired testing coverage in addition to satisfying the test criteria. Quite often, additional tests to achieve increased coverage are derived from the initial functional tests.

Executing the Functional Tests. Before processing with an AVS, each of the initial functional tests should be used to exercise the software and the output should be evaluated against the test criteria. This is important for two reasons:

- It provides a baseline set of output from the uninstrumented software for later comparison to that from instrumented software.
- It demonstrates the ability of the test team to prepare test data, execute the program, and interpret results.

This step requires that the program be complete, that source code be compiled error-free, and that test data result in acceptable execution (though not necessarily expected behavior).

The next step is to execute each of the functional tests with instrumented software and determine initial coverage. This necessitates AVS processing to build the library (BASIC and STRUCTURAL),* instrument appropriate portions of the software (INSTRUMENT),* compile and execute the instrumented software with the AVS probe routines (Test Execution),* and obtain coverage reports (ANALYZER).* It is very important that normal program output from Test Execution be checked against the baseline output. If discrepancies exist between the two, this is direct evidence that the addition of instrumentation, in some way, has exposed a software malfunction. Some of the reasons for this type of error are:

- The software test object is sensitive to time or space perturbations (i.e., it is not suitable for AVS testing).
- The probe routines were improperly added (e.g., placed in the wrong overlay link).
- The test data or test environment is different from that of the baseline test.
- The computation process has caused the malfunction (e.g., using the wrong COMPOOL to compile; inconsistent code generated by the compiler).
- Computer resources are inadequate to process instrumented code (e.g., compiler limitations regarding number of external symbols, or memory capacity for expanded code).

The AVS capability for evaluating test effectiveness (ANALYZER) provides a detailed and comprehensive analysis of testing coverage. Reports are generated on execution tracing, module and path coverage, timing, and modules and paths not exercised. For the initial functional tests this information should be evaluated in some detail, since it represents the point of departure for subsequent testing. Since the AVS only assists the test team in preparing the software for Test Execution, the initial coverage results may not accurately reflect actual coverage. For example, in JAVS the tester selects the placement of a "beginning of test" signal and an "end of test" signal to the probe routines, and they record coverage only during that interval of execution. Thus the coverage reports are limited to code executed within this interval. Some reasons for unexpected coverage results are:

- The functional test does not exercise the expected modules at all.
- The selection of test point placement is improper.
- The selection of modules to instrument is not compatible with the functional test, perhaps indicating erroneous definition of system structure.

If anomalous results occur at this point it is important to review decisions made during the previous steps in the testing process before proceeding.

---

*JAVS processing keywords; see Sec. 3.3.

The next step is to execute each of the functional tests with instrumented software and determine initial coverage. This necessitates AVS processing to build the library (BASIC and STRUCTURAL),* instrument appropriate portions of the software (INSTRUMENT),* compile and execute the instrumented software with the AVS probe routines (Test Execution),* and obtain coverage reports (ANALYZER).* It is very important that normal program output from Test Execution be checked against the baseline output. If discrepancies exist between the two, this is direct evidence that the addition of instrumentation, in some way, has exposed a software malfunction. Some of the reasons for this type of error are:

- The software test object is sensitive to time or space perturbations (i.e., it is not suitable for AVS testing).
- The probe routines were improperly added (e.g., placed in the wrong overlay link).
- The test data or test environment is different from that of the baseline test.
- The computation process has caused the malfunction (e.g., using the wrong COMPOOL to compile; inconsistent code generated by the compiler).
- Computer resources are inadequate to process instrumented code (e.g., compiler limitations regarding number of external symbols, or memory capacity for expanded code).

The AVS capability for evaluating test effectiveness (ANALYZER) provides a detailed and comprehensive analysis of testing coverage. Reports are generated on execution tracing, module and path coverage, timing, and modules and paths not exercised. For the initial functional tests this information should be evaluated in some detail, since it represents the point of departure for subsequent testing. Since the AVS only assists the test team in preparing the software for Test Execution, the initial coverage results may not accurately reflect actual coverage. For example, in JAVS the tester selects the placement of a "beginning of test" signal and an "end of test" signal to the probe routines, and they record coverage only during that interval of execution. Thus the coverage reports are limited to code executed within this interval. Some reasons for unexpected coverage results are:

- The functional test does not exercise the expected modules at all.
- The selection of test point placement is improper.
- The selection of modules to instrument is not compatible with the functional test, perhaps indicating erroneous definition of system structure.

If anomalous results occur at this point it is important to review decisions made during the previous steps in the testing process before proceeding.

---

*JAVS processing keywords; see Sec. 3.3.

Structure-Based Testing. Once the test team is confident of the quality achieved in obtaining initial functional test results, testing proceeds with AVS assistance as follows:

- Selecting a testing target from information in AVS coverage reports for previous tests
- Constructing new tests to improve coverage using AVS retesting assistance
- Performing Test Execution with new tests capturing software behavior data
- Analyzing test results from AVS coverage reports

These steps are repeated until the test coverage objectives have been met.

Software testing can be performed at the single-module or system level. At the single-module level, the retesting target is a set of DD-paths which have not been exercised. At the system level, retesting involves identification of target modules with low coverage and analysis of intermodule dependencies. Single-module testing may be viewed as a part of system-wide testing.

In order to construct new test cases, information about the control structure of the module and its input domain is used. The relationship of the module's input domain to the system input data must be determined in order to test the module in its normal environment (i.e., its position in the invocation structure of the software system). This may prove difficult, or not even possible, since communication paths to the module may be blocked (e.g., by protective code or by lack of knowledge as in FMIS testing). If this is the case, a special test environment may be needed to thoroughly exercise the module.

Systematic Single-Module Testing

The testing process for single-module coverage has a single objective: to construct test cases which cause execution of as yet unexecuted DD-paths within the program. Testing is over when all DD-paths have been exercised or when those which have not been exercised are shown by the program tester to be logically unexecutable.

Two questions arise in addressing the task of single-module retesting: what are the targets for retesting? and, once selected, what assistance is available to exercise the targets?

The DD-path selection criteria should attempt to maximize collateral testing; i.e., to exercise more than one unexercised DD-path with each new test case. Several guidelines can be used to aid the selection:

1. In a cluster of unexercised paths, choose an untested DD-path that is on the highest possible control nesting level. (JAVS statement and DD-path coverage reports print the nesting level;

see Fig. 4.3 or Fig. 4.12). This selection assures a high degree of collateral testing, since some of the DD-paths leading to and from the target must be executed.

2. A reaching set is a sequence of DD-paths which lead to a specified DD-path. At user request, JAVS determines reaching sets to include or exclude iteration. Choose a DD-path which is at the end of a long reaching set as the target (see Fig. 4.7, statement 32). In addition to collateral testing, it is likely that the resulting test case input will be similar to data which resembles the functional nature of the program.

3. If a prior test case carries the program near one of the untested DD-paths, it may be more economical to determine how that test case can be modified to execute the unexercised path. For example, JAVS post-test analysis reports (see ANALYZER,HIT and ANALYZER, DDPTRACE in Ref. 5) show which DD-paths were executed during each separate test case.

4. If the analyses required for a particular DD-path selection are difficult, then choose a path which lies along the lower-level portions of its reaching set. This can simplify the analysis problem.

5. Analyze the untested DD-path predicate (conditional formula) in the reaching set for "key" variable names which may lead directly to the input.

6. Choose DD-paths whose predicates evaluate functional boundaries or extreme conditions; exercising these paths frequently uncovers program errors.

Most of these guidelines depend upon the identification of DD-path reaching sets. One of the tasks performed by the JAVS testing assistance processor (ASSIST) is the determination of these sets. The user inputs the desired path number to be reached, and ASSIST generates the reaching set of paths from the module entry or from a designated starting path to the specified path. For the generated set of paths, the key program statements are printed, including the necessary outcome of any conditional statements which are essential. The user may specify iterative or noniterative reaching sets to be generated.

The process of relating paths which are targets for retesting to the input for generating new test cases is highly dependent upon the design of the test program. JAVS shows what code segments have not been exercised by the data (Figs. 4.2 and 4.11 - 4.13) and the program paths that lead up to any selected DD-path target (Fig. 4.7). The tester must analyze the predicates in the testing targets for important variables which may be described in program comments or documentation. These variables can be traced throughout the program by using the JAVS cross reference report (Fig. 4.8) and the module interdependency (Fig. 4.10) and invocation parameter reports (if the variables are passed as parameters) (Fig. 4.9).

When the new set of test cases is generated, it can be added to the previously input data or executed alone by the instrumented modules. The results of this Test Execution are then processed by the post-test analyzer to see if the coverage is satisfactory. At this time, the user may see problems in the software which require code changes. The JAVS documentation reports can be used to determine the effects that the code modification will have within a single module or within the data base system of modules.

Systematic System-Wide Testing

The system testing effort can be organized according to two fundamentally distinct strategies: (1) bottom-up system testing, and (2) top-down system testing.

Bottom-Up Testing: This testing strategy attempts to provide comprehensive system testing coverage by building test cases from the bottom of the system invocation hierarchy first, and extending these test cases upward during the continuing and concluding testing phases. Bottom-up testing may require the use of special testing environments (see below), but is likely to achieve the best overall testing coverage.

Top-Down Testing: This testing strategy deals with an entire software system first, and, after subsystem (or component) effectiveness is measured, proceeds downward through the software system's invocation structure. Test case data is added only at the topmost level and, as a result, a set of system-wide test cases are developed directly.

The optimum system testing strategy for a particular system generally combines the two strategies. The choice is based on the level of coverage achieved, the difficulty of proceeding upward or downward in the system organization, and the effort required to establish a testing environment in each case.

The basic ingredients of the systematic software system testing methodology are the following:

- The ability to perform comprehensive single-module testing for each invokable module
- Knowledge of the system's invocation structure
- Previous (and initial) system testing coverage measures
- A next-testing-target selection function to allocate testing effort

The general form of the system testing methodology is shown in Fig. 5.5, which emphasizes continuous use of a system testing coverage measure. The interaction between the system testing coverage measure and the process of selective application of the single-module testing methodology is described next. The coverage value can also be used to select the best next testing target.

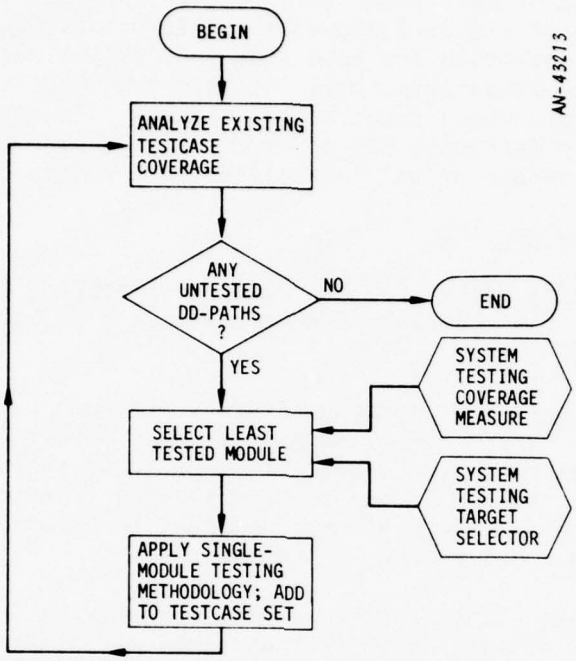


Figure 5.5. *Overview of System Testing Methodology*

System-wide testing coverage can be measured in terms of the coverage for each module, or in terms of the coverage for an identifiable subset of related modules (i.e., a component). The coverage measure can be used to select the best next testing target.

The simple per-module coverage measure will direct testing effort toward the module which is the least tested. The per-element form of composite testing coverage allocates testing effort toward the component which has undergone the least testing.

The measure actually used should depend on the internal structure, and possibly the functional requirements, of the software system as a whole. The measure should unambiguously identify the module(s) least tested, but should tend to identify a number of possible testing targets. The choice between them should be made within the confines of the invocation hierarchy, and by considering the two important variations of testing stategy: top-down testing, and bottom-up testing.

Bottom-Up Testing

In bottom-up testing, a system tester has two choices: where to concentrate the testing effort, and where to provide test case data.

1. Test case data can be supplied through the existing data input points. The system tester must be aware of data transformations

performed prior to delivery to the module on which he is currently working. These transformations may make it difficult, or impossible, to exercise the current testing target.

2. The test case data can be supplied through a separate testing environment designed and implemented specifically to provide for testing of a single system element. At the system testing level the testing would be performed with the normal data input mechanism.

The choice between these two mechanisms for providing test case data must be based on the specific internal features of the software system.

The AVS intermodule dependencies and symbol cross reference capabilities can be used along with module input and output information to select the appropriate technique.

Figure 5.6 shows the use of a bottom-up strategy to select a testing target. The selector emphasizes comprehensive testing of single modules before components, components before subsystems, etc. The selection rule is the following:

> Bottom-Up Testing Procedure. Begin the testing effort with modules which are invoked at the end (i.e., the lowest level) of the invocation chain. Advance upward in the hierarchy only after all terminal-branch elements have been exercised comprehensively. The testing target is always the least tested module that is furthest down in the invocation hierarchy.

It may be necessary to accept less-than-full exercise of each module as a reasonable testing strategy. This amounts to assigning a minimum threshold of testedness for each module. Bottom-up testing will assure that all possible single-module testing will have occurred first; the technique has a high likelihood of transmitting this high level of exercise to the topmost levels of the software system. Special testing environments may be required along the way, however.

Top-Down Testing

For top-down system testing, the testing environment at each stage is the obvious one: the topmost element of the software system will control some data and will selectively pass it downward in the invocation structure to the subsystems, to the components, and, eventually, to individual modules. New test case data is added at the points where the software system normally accepts input data. These normal data input points are not necessarily part of the topmost program; there may be special "data entry" subsystems or components which are invoked by the topmost program specifically for this purpose.

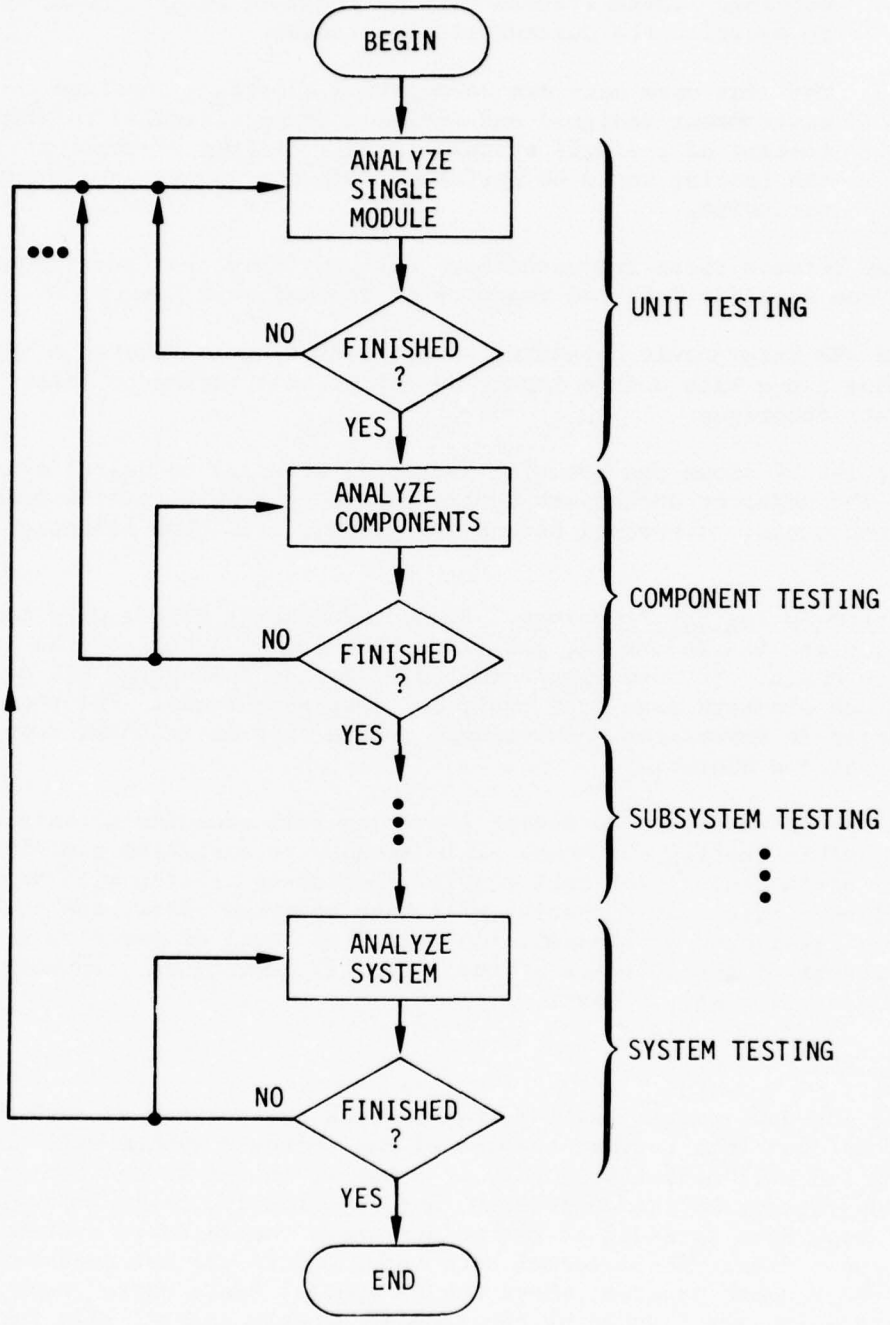


Figure 5.6. System Testing Methodology (Bottom-Up)

Top-down testing is almost the reverse of bottom-up testing, and involves selecting for further testing effort the largest elements of the system (i.e., modules which control entire subsystems or components) before attempting testing of modules at the lower levels. The selection rule that corresponds to this testing strategy is:

> Top-Down Testing Procedure. Begin testing at the beginning of the invocation structure (i.e., at the highest level). Advance downward only after the highest-level modules have been comprehensively exercised (to a preset threshold). The testing target selected is always the least tested module which resides at the level in the invocation hierarchy at which testing is currently proceeding.

This selector operates strictly in terms of the chain depth within the invocation structure tree. It is generally undesirable to accept less than 100% testing coverage at any stage of the process, however, since doing so may mean that an important invocation chain is missed. That is, any invocation chain which is the only one that permits testing some lower-level module must not be skipped in the early levels of testing.

The top-down method generally assures maximum collateral testing prior to attacking any particular lower-level module. Stubbing of system elements may be necessary to conserve limited testing resources, however.

General Strategy

The best approach for systematically testing a large software system will depend on the specifics of that system's elements; it is not possible to state a universally applicable strategy. Mixtures of the top-down and bottom-up approaches may well cost the least, and may result in the greatest testing coverage.

The results of a test activity depend to a large extent on the capability and ingenuity of the test team. An AVS does offer tools, not previously available, to make testing more effective. Application of those tools to particular situations is the responsibility of the testers. There are, however, some guidelines for selecting the most appropriate AVS capabilities for particular situations. Some of these are described below.

> Incomplete documentation. Use AVS resources to build the library (BASIC and STRUCTURAL) and obtain documentation reports on the software (DOCUMENT). Construct missing documentation needed to start testing.

> Incomplete software. Use AVS resources to build the library (BASIC and STRUCTURAL) and obtain documentation reports on the software (DOCUMENT). To complete the test environment, first identify top-level modules and external library dependencies (Fig. 5.7). Next, construct a test driver for each top-level module: use AVS module invocation definition and cross reference for calling protocol and input domain. Provide stubs (i.e., dummy modules) for externals which are referenced but are not present on system or auxiliary libraries; use AVS module

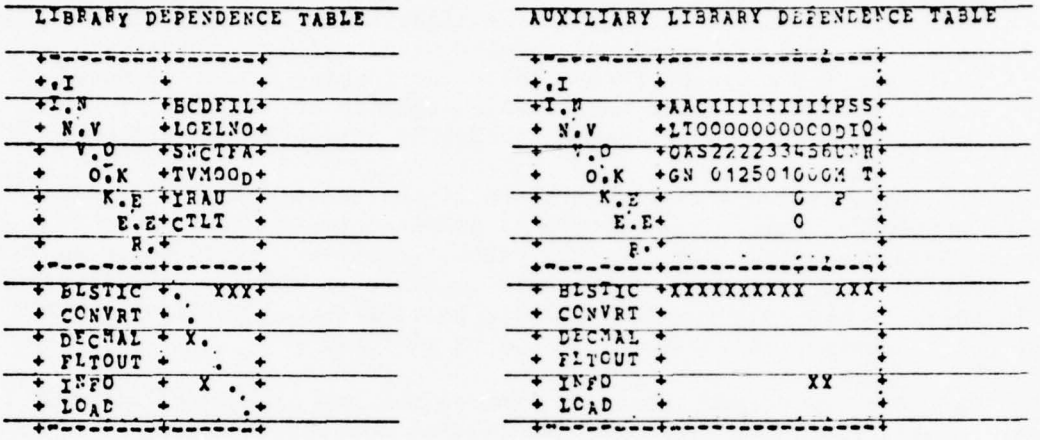


Figure 5.7. Tables Showing Interdependencies. These tables show relations between all modules on the data base library and between library modules and externals not on the data base library.

---

invocation references for calling protocol. Documentation supplied with the software should be consulted for module interface specifications.

Single-Module Testing. Use unexercised DD-paths report (ANALYZER, NOTHIT) to identify potential test target paths. Use module listing (PRINT, MODULE) or DD-path definitions report (PRINT, DDPATHS) to find nesting level of unexercised paths. Use control flow picture (ASSIST, PICTURE) for overview of module structure. Select testing target from reaching set for target path (ASSIST, REACHING SET) to determine module inputs which will cause target path to execute. Generate test data (see below).

System-Wide Testing. Use module coverage summary and DD-path coverage summary to identify potential test target modules. For any module never invoked use inter-module dependencies (DOCUMENT) to determine what modules invoke it directly, and indirectly through other modules. Identify which higher-level modules were executed. Using cross-reference reports together with intermodule dependence reports, identify what modules affect invocation. Modify test data to cause invocation of target module. Several cycles of top-down testing may be required if the unexercised inter-module control structure is at all complex. Apply single-module testing techniques to increase intra-module coverage.

Unknown behavior. Plant document probes (PROBD) to capture imbedded description information in test execution trace. Examine test trace report to identify behavior with probe location. Use results to select appropriate test points.

Unexpected behavior. Use AVS assertion statements (e.g., JAVS execution descriptions) to isolate causes. At the beginning of the module, insert assertion statements for expected condition of module inputs. At the end of the module where control is returned, insert assertion statements for expected conditions of module outputs. At intermediate locations in the module, insert assertion statements for expected conditions of module behavior. Examine Test Execution output for report of unexpected behavior.

Testing Assessment

At the conclusion of the testing an assessment should be made of the results. This summary should include the following:

- Documentation of the methods and extent of testing: strategy for testing, coverage achieved, test cases used, dynamic behavior modes tested, and identification of logically unreachable code

- Determination of the consistency between the software and its functional specifications: what specific functions are implemented, what unspecified functions are implemented, what restrictions not specified are embedded

- Evaluation of existing software documentation: errors, inconsistencies with implemented software, missing information, superfluous information

There are several side benefits to be realized from testing. For example, the software can be optimized by removal of unreachable code or code which implements extraneous functions. AVS documentation reports are useful here in determining the extent of changes to the software (e.g., modules and data structures affected) and the amount of retesting necessary after software modifications have been made.

6 ADVANCED AVS CAPABILITIES

Prior to the availability of Automated Verification Systems, software developers of necessity relied on extensive manual testing to demonstrate software performance. The information about software characteristics was also largely manually composed, supplemented by meager reports generated as side products from compilers, loaders, and other system software. With access to an AVS with current capabilities such as JAVS or RXVP, the situation changes dramatically: software developers can expect to test software systematically with support from specially designed test tools, to document software automatically at various stages in its development, and to build in quality during development by utilizing AVS features which perform both static and dynamic tests of software quality. Testing experience with JAVS (see Sec. 4) has demonstrated some direct benefits of using an AVS on existing software, not only in achieving comprehensive software testing but also in generating high-quality, accurate software documentation. The indirect benefits of these tests included illumination of software properties which are difficult to test, and identification of extraneous code and unused data structures.

More advanced Automated Verification Systems can be expected not only to automate additional test, documentation, and quality-checking services for existing software but also to improve the quality of new software by other mechanisms such as tool-supported language extensions. To achieve full benefit from the availability of an AVS, new software should be designed with the use of the AVS as one of the considerations. Maintenance and testing activities provide opportunities to insert in existing software additional information analyzed by an AVS (e.g., assertion statements on program performance).

This section presents a forecast of the capabilities offered by an advanced Automated Verification System. Current AVS capabilities emphasize the testing of existing software; future AVS capabilities will include tools for analyzing both existing software and new software which is designed to exploit the AVS. Furthermore, developments in languages, system software, application software, and hardware design which are independent of AVS considerations will most certainly influence the direction of advanced AVS development.

6.1 CURRENT AVS IMPLEMENTATION

Today's AVS is applicable to today's software. The typical software system subjected to an AVS (1) already exists, (2) is implemented in a popular procedural language such as FORTRAN, JOVIAL, or PASCAL, (3) most likely was not developed with top-down, bottom-up, or structured programming development techniques, (4) is not well documented for ease in testing and maintenance, and (5) was not designed to exploit AVS capabilities. The currently operational AVS is an experimental tool, incorporates state-of-the-art analysis techniques, operates independently of other software tools (e.g., compilers), and is not yet a primary tool in the software development cycle as are compilers, linkage editors, and operating systems.

So far, the use of the AVS has been limited and experimental. The objectives of the current effort are primarily concerned with gaining experience in using the AVS, developing techniques for using it effectively, measuring its performance, and establishing the extent of its applicability to current systems. The limited testing experience described in Sec. 4 utilized only a part of the JAVS capabilities. None of the software test objects was designed to use an AVS in testing; furthermore none was implemented with language extensions (i.e., executable assertions) or with the aid of documentation tools provided by JAVS. In addition, the test teams were largely inexperienced in using an AVS, although they included individuals who had developed several AVSs. In spite of this, the effectiveness of JAVS in testing and documenting both large and small software systems was demonstrated. For example, the high statement coverage required in the JAVS software acceptance tests was quickly accomplished and certified with AVS support for instrumentation and test coverage analysis. To do the same task manually on the JAVS software would be impractical.

In documenting the JAVS software, the task of producing the required documentation, to the level of detail called for, was greatly simplified by using the AVS to produce all the structural information. Only the descriptive semantic and organizational details were manually prepared.

6.2 FUTURE AVS CAPABILITIES

Compared to today's AVS, tomorrow's AVS will be a far more effective tool for the software developer. It will offer exapnded capabilities and will be easy to use and cost-effective. The developer will design and implement software to take advantage of the AVS, in much the same way that current software is designed and implemented to take advantage of current tools: high-level compilers, system loaders, libraries, text maintenance tools, and operating system processors. Existing languages will be extended and new languages developed to improve software quality. Some of these language features will supply additional information for analysis by the AVS together with the (essential) procedural information.

In the following subsections, the capabilities to be found in advanced AVS tools are described. They fall into two main categories: static tools, which analyze the properties of the software without executing it, and dynamic tools, which are associated with executing the software. Where appropriate, current AVS capabilities are indicated, as are those which depend on language extensions.

6.2.1 Static Tools

Static AVS tools analyze software for its static properties, i.e., without executing the software. Some tools are concerned with a small part (e.g., modules) of the software system while other analyze subsystems (e.g., functionally related sets of modules) or even the entire software system. They provide such services as:

- Syntactic Documentation
- Coding Standards Checking
- Control Structure Analysis
- Consistency Checking
- Program Proof of Correctness

Syntactic Documentation. Comprehensive syntactic documentation of software systems is essential to software and testing maintenance. The most common forms are source listings, symbol tables, and cross-reference lists. Typically AVS tools enhance the software source listings with automatic control-structure indentation and nesting-level indicators. Symbol tables identify symbols used and symbol specifications (e.g., type, precision, dimensions, scope). Cross-reference lists identify where (e.g., by module or by statement) and how (e.g., declared, set, used, invoked) symbols are referenced.

Since some compilers operate on separate compilation units (e.g., the GCOS JOCIT JOVIAL compiler accepts only a single START-TERM sequence), it is not possible to obtain a cross-reference listing over a selected set of modules directly from the compiler. An AVS with access to the complete source text of the software generates this information readily. The designated set of modules may be the complete software system, a functionally related subsystem, those modules resident in a single memory load, those on the same chain in an invocation hierarchy, or other suitable combinations.

In addition to the above, current AVS tools offer more extensive reports such as:

- Explicit module invocation matrix, showing direct invocation of each module to all other modules, whether or not known to the AVS
- Inter-module invocation tree, showing each module's position in the invocation hierarchy
- Module invocation environment, showing the local invocation hierarchy of the designated module (i.e., other modules which call the designated module or are called by it, both directly and indirectly, for a specified number of levels)
- Module invocation source text, showing formal invocation declaration of the designated module, all actual invocations of the designated module known to the AVS, and all invocations from the designated module.

Examples of these reports are contained in Ref. 5.

The recent experience with AVS applications has indicated other useful syntactic documentations attainable with existing technology:

- Explicit and implicit module invocation matrix, showing both direct and (possible) indirect invocation of each module to other modules, whether or not known to the AVS.
- Scope data definition structure, showing scope hierarchy data definition (e.g., as determined by module nesting structure).
- Symbol reference matrix, showing symbols explicitly referenced by module.
- Symbol access matrix, showing accessibility of each symbol to modules and indicating those symbols which are (1) explicitly referenced and (2) implicitly accessed through actual invocation parameter lists (i.e., the module communication space).
- Symbol control structure reference matrix, showing symbols explicitly referenced in control statements (e.g., IF, GOTO, SWITCH).
- File symbols reference matrix, showing I/O interfaces.
- Symbol file data transfers, showing symbols referenced in I/O data transfers.
- Symbol-to-symbol relationships, showing symbols directly related to another (e.g., by data structure definition such as table membership or overlay, or by usage in some executable statement).

Although none of the above actually requires extensions to the language, some mechanism to designate a module's membership in one or more selected sets would simplify the use of the AVS. (Current AVS tools such as JAVS provide for selecting a list of modules by name from the known modules under user command.)

Coding Standards Checking. The enforcement of coding standards is largely a preventive measure used to avoid problems often associated with poor language usage. Standards checkers analyze code for conformance and identify offending statements. Standards usually are imposed on module size (number of executable statements), permitted language constructs (language subset), comments (all control statements preceded by descriptive commentary), and symbol naming conventions (all modules have long names, mnemonic data symbols, increasing sequential order for statement labels). A common technique for enforcing standards is to use a standards checker as a preprocessor to the compiler and prevent nonconforming modules from being compiled. With an AVS, however, standards checking can be integrated along with other static analysis tools, such as those described below.

Control Structure Analysis. The identification of the control structure embodied in software is necessary to other AVS functions. Structural analysis determines the possible program control flow by identifying the sequential code segments which define decision-to-decision paths (DD-paths) within each module and by locating all inter-module invocations and control transfers. This information is the basis of static analysis of intermodule structures, testing coverage analysis and assistance, data flow analysis, and other analyses which require knowledge about program flow. As part of its function, the structural analysis processor (currently part of AVS tools) includes checks on the control structure of the software such as identification of:

- Structurally non-terminating module
- Structurally unreachable code
- Direct transfers into a deeper or alternate control nesting level ("borrowed code")
- Direct out-of-module transfers within the module invocation hierarchy ("escape" path)
- Direct out-of-module transfers to another chain in the module invocation hierarchy ("threaded" path)
- Circular module invocation (recursion)

These constructs, although permitted in the language, may not (or cannot) be detected by normal compiler operation. Other structural services currently provided in existing AVS implementations[5,10,11] include:

- Reaching set and reaching sequence analysis for a specified target statement or path.
- Program restructuring to a standard form (i.e., single-entry single-exit structured modules).

Current AVS structural analysis is based on algorithms for non-recursive, non-concurrent software written in conventional procedural languages. Advanced AVS structural analysis must cope with recursion (JOVIAL/J73), concurrency (Concurrent PASCAL), and advanced control structures which support new language features such as software fault tolerance (see Sec. 6.2.2), escape mechanisms, and co-routines.

Consistency Checking. Although most languages require software to be internally consistent, many compilers do not (or cannot) check for consistency. Currently an AVS with access to the entire software source code can perform valuable checks for consistency such as:

- Agreement between actual and formal parameter lists for number of arguments and argument specifications
- Consistent use of syntactic types (e.g., no implied type conversion, legal operations, proper number of subscripts)

structures). It is generally accepted that simplification by the application of a set of rules is the best approach to take. With the current state of the art, interactive verification condition generation and simplification appears to be most promising.[12]

6.2.2 Dynamic Tools

Dynamic AVS tools support the analysis of execution characteristics of a program (i.e., the dynamic properties of program behavior). There are two general categories of dynamic tools: those which are concerned with testing software and those which support software fault tolerance.

Testing. Current AVS implementations have emphasized the testing of software to achieve higher quality. There is a wide variety of testing analysis tools which can be used on existing software such as:

- Control structure instrumentation which records the dynamic flow of program control through insertion of probes
- Coverage analysis which reports the program paths exercised (or not exercised) during testing in varying levels of detail
- Performance analysis which reports timing of modules.

Executable assertions about program behavior may also be inserted in existing software. These result in instrumentation probes which report violations of the assertions during test execution.

Advanced AVS implementations will offer not only extensions of these types of analysis but also tools which support test environment generation and test history maintenance. Dynamic consistency checking is one type of expanded analysis capability which can be used to expose errors not found with static consistency checks. These dynamic checks include detection of violations of:

- Subscript limitations
- Declared variable ranges
- Timing constraints
- Set-before-use and use-after-set practices

An important capability for advanced AVS implementation is automated test environment generation. With current AVS capabilities the tester must construct the test environment manually. He assembles the code to be tested from instrumented and uninstrumented modules, sometimes producing new software for test drivers and module stubs to make a complete program. Test data is also manually prepared. He may use reports from the AVS to identify what modules are necessary, what interfaces must be supplied, and what variables are directly affected by test data inputs.

In this respect the AVS is a passive participant in the testing activity. A more active role for the AVS in automated test driver and stub generation and test data generation is certainly desirable. For example, data flow analysis coupled with use of module input and output declarations and entry and exit assertions can provide the basis for automated test environment support. For a target module, a test driver can be constructed which contains a synthesis of the module's communication space, module invocations, and code to initialize input data; stubs for other modules invoked by the target module can be similarly constructed. The tester then supplies the required data. Entry assertions verify the data; exit assertions verify the test results.

Systematic software testing often requires keeping careful records of the testing history of software systems: what tests were done, what test data was used, what was accomplished. This information is used as a test management device to select test strategies which minimize the amount of additional testing. If the software is modified, previous test records can be used to select test data and software configurations for retesting activities. Current AVS implementations permit the tester to identify each test, but the AVS makes no direct use of the information. Record keeping is largely a clerical process, and, if manually done, it is prone to error. Although automated test history maintenance can involve retention of large quantities of information (e.g., detailed coverage data), some automated mechanism for identifying, extracting, and retaining essential data is desirable for advanced AVS implementation.

Software Fault Tolerance. Another approach to improving software quality is to provide a mechanism whereby software can continue to function properly although some failure (hardware, software, or bad data) has occurred. Fault tolerance attempts to increase the reliability of a system through dynamic redundancy. A fault tolerant software system requires redundant software or hardware in order to:

1. Detect faults before major damage occurs

2. Diagnose faults so repair may be performed

3. Recover from faults by re-establishing an acceptable system state.

Correctness proofs assume that no hardware faults will occur and that only specified data inputs will be processed. For this reason "correct" systems may not be reliable systems. Fault tolerance techniques assume that hardware faults will occur, data input specifications will be violated, and software errors are inevitable. A variety of approaches for attaining fault tolerance in software are currently being investigated.[12] All use various combinations of software and/or hardware for support. This has strong implications for the role of AVS in testing: some mechanism must be provided to simulate faults in order to test error detection, repair, and recovery functions.

6.2.3 New AVS Requirements

There are a number of ongoing developments in the computer arena which impact the requirements for AVS capabilities. Among these are

- New software languages and software designs (e.g., to support concurrency, fault tolerance, restricted data access).
- Non-conventional hardware architectures (e.g., tagged architecture, distributed data processing).
- System software which supports new developments in hardware and application software.

All of these factors will impose new requirements on AVS facilities.

Current AVS implementations have been designed for a stable environment of rather conventional characteristics: e.g., a large host computer, with ample auxiliary storage for a substantial data base, and a computer operating system supporting large program applications. This has been an adequate assumption for experimental AVS usage. The need for AVS adaptation to more restricted host environments (e.g., to mini-computers) has now become evident. This impacts the design of AVS software itself. For example, in more restrictive environments, each AVS tool could be separately implemented for the most efficient use of computer resources.

Many AVS functions closely parallel those of current compilers: for example syntactic source analysis. An AVS developed independently of a compiler does offer an independent check on these functions. This, however, is also a drawback since erroneous conflicts in interpretations of source text between the AVS and the compiler are a source of difficulty in AVS usage. It is not at all unreasonable to expect advanced AVS implementations to make use of compiler source text analysis functions in constructing portions of the database. This can be accomplished by requiring the compiler to retain the information in an acceptable form for input to an AVS.

APPENDIX A

GLOSSARY OF AVS TERMINOLOGY

AVS. Automated Verification System.

AVS Database. In an AVS, the collection of information maintained internal to to the AVS and which contains all pertinent data about all modules known to the AVS.

Actual Parameter. In an invocation of a module, the set of variable names passed to the invoked module in the actual parameter list.

Automated Verification System. A system for the analysis of software systems oriented toward systematic, comprehensive testing (exercise) as a means to perform software verification. JAVS is an example of such a system.

Bottom-up Testing Strategy. A systematic testing philosophy which seeks to test those modules at the bottom of the invocation structure earliest.

Collateral Testing. Collateral testing is that testing coverage which is achieved indirectly, rather than as the direct object of a testcase activity.

Communication Space. The communication space of a module consists of those symbols, known within the module, by which information can be passed to or from the module. Communication space mechanisms consist of formal parameters, global variables, and return parameters.

Computation Directive Instrumentation. The process of producing an altered version of a module as the result of user-inserted directives which is logically equivalent to the unmodified module, but which contains executable code that provides for collecting information about the dynamic behavior of the module during its execution.

Computation Structure. The operation of the program on the data.

Control Level. (Control nesting level.) The control structure of a program is hierarchical. At the module's entry, the control level is 0. Each new DD-path or beginning of a BEGIN-END block increases the control level; the end of each DD-path or BEGIN-END block decreases the control level.

Control Structure. The set of program statements which alter the normal sequential flow from one statement to the next.

Data Structure. Organization of the data on which the program operates.

DD-Path. A DD-path, or decision-to-decision path, is the set of statements in a module which are executed as the result of the evaluation of some predicate (conditional) within the module. The DD-path should be thought of as including the sensing of the outcome of a conditional operation and the subsequent executions up to, and including, the computation of the next predicate value but not including its evaluation.

DD-Path Instrumentation. The process of producing an altered version of a module which is logically equivalent to the unmodified module but which contains calls to a special data collection subroutine which accepts information as to the specific DD-path sequence incurred in an invocation of the module.

DD-Path Predicate. A logical formula involving variables/constants known to a module and, possibly, the values .TRUE. and .FALSE., which must be satisfied for the DD-path to be executed.

Decision Statement. A decision statement in a module is one in which an evaluation of some predicate is made which (potentially) affects the subsequent execution behavior of the module.

Decision-to-Decision Path. See DD-path.

Directive. A user-supplied statement imbedded in the source text which directs the AVS to perform a specified function.

Essential DD-path. A DD-path in a reaching set which must be executed in order to reach the designated DD-path.

Executable Statement. A statement in a module which is executable in the sense that it produces object code instructions.

External Label. See global label.

Flow. A particular sequence of DD-paths.

Formal Parameter. For an invokable element of program text, the set of variable names which are assigned value outside of the program text.

Functional Specifications. A set of behavior and performance requirements which, in aggregate, determine the functional properties of a software system.

Functional Test Cases. A set of test case datasets for software which are derived for testing specific tasks or functions.

Global Label. A statement label residing in one module but which is transferred to from other modules.

Global Variable. In a module, a global variable is one which may receive a value as the result of actions outside the module.

Input Domain. See input space.

Input Space. The input space of a module consists of that subset of a module's communication space which can be (1) altered externally to the module, and (2) which is (potentially) used within the module in a way that affects its execution.

Instrumentation. The automatic insertion of software probes (e.g., invocations to data collection routines) to capture information during execution.

Intermodule Dependencies. Generally refers to module interaction via invocations but can also include interaction due to external label transfers.

Invocation Point. The invocation point of a module is the first statement in the module (in JOVIAL, a program, procedure, or close), or, if the module has multiple entry points, an entry statement.

Invocation Structure. The hierachy of invocations of one module by another within a software system.

Iterative Flow. Iterative flow is represented by a sequence of DD-paths with the property that some DD-path belonging to the sequence can be executed one or more times.

JOCIT JOVIAL. (JOVIAL Compiler Implementation Tool). A dialect of JOVIAL/J3.

JOVIAL. Unless further specified, any of the dialects of the family of JOVIAL languages.

JOVIAL/J3. The JOVIAL programming language, J3 subset, as defined in Air Force Manual AFM-100-24.

Memory. A module is said to have memory if there is some interior code condition which makes it possible to execute some DD-path only by making two or more invocations of the module.

Memory Space. The memory space for a module consists of those cells known to the module which allow it to have memory (see Memory).

Module. A module is a separately invokable element of a software system.

Non-Executable Statement. A declaration or directive within a module which does not produce (during compilation) object code instructions directly.

Output Space. The output space of a module consists of the collection of variables, including file actions, which are (or could be) modified by some invocation of the module.

Path. A sequence of DD-paths.

Predicate. A formula involving variables/constants and relational operators, which can be evaluated to .TRUE. or .FALSE..

Program Validation. The process of developing and verifying the correspondence between an implemented software system and the set of functional specifications which correspond to it.

Program Verification. The process of verifying that a set of functional test cases meets structural testing goals.

Reaching Set. The set of all DD-paths that connect together to form paths from one designated DD-path to another.

Software System. A collection of modules, possibly organized into components and subsystems, which solves some problem.

Software Validation. See Program Validation.

Structural Instrumentation. Instrumentation of module invocation entries, exits and DD-paths without changing the logic of the uninstrumented source code.

Structural Test Cases. A set of test case patterns, derived from the control structure of a module (or a collection of modules). The combination of a structural test case and appropriate program input data results in a functional test case.

Test. A test is one or more unit tests of one or more modules.

Test Case. See Test.

Test Case Dataset. A test case dataset is a specific set of values for variables in the communication space of a module which are used in a test.

Testing Coverage Measure. A measure of the testing coverage achieved as the result of one unit test, usually expressed as a percentage of the number of DD-paths within a module which were traversed in the test.

Test Execution. Execution of the test object in which one or more modules of the test object have been instrumented. Output differs from normal execution output in that information captured from the instrumented code is written to a sequential file for later analysis.

Test Object. The software to be tested.

Testing Stub. A testing stub is a module which simulates the operations of a module which is invoked within a test. The testing stub can replace the real module for testing purposes.

Testing Target. The current module (system testing) or the current DD-path (unit testing) upon which testing effort is focused.

Testing Verification. See program verification.

Top-down Testing Strategy. A systematic testing philosophy which seeks to test those modules at the top of the invocation structure earliest.

Unit Test. A unit test of a single module consists of (1) a collection of settings for the input space of the module, and (2) exactly one invocation of the module. A unit test may or may not include the effect of other modules which are invoked by the module undergoing testing.

Unreachability. A statement (or DD-path) is unreachable if there is no logically obtainable set of input-space settings which can cause the statement (or DD-path) to be traversed.

REFERENCES

1. Revised Statement of Work for Automated Verification System, PR B-3-3209, Rome Air Development Center, Griffiss Air Force Base, New York, April 16, 1973.

2. E. F. Miller, Jr., Methodology for Comprehensive Software Testing, General Research Corporation CR-1-465, June 1975. Available as RADC-TR-75-161, Interim Report, June 1975 at Rome Air Development Center, Griffiss Air Force Base, New York.

3. Standard Computer Programming Language for Air Force Command and Control Systems, Department of the Air Force AFM 100-24, June 15, 1967.

4. C. Gannon, N. B. Brooks, R. J. Urban, JAVS Technical Report: RADC-TR-77-126, Volume I, "User's Guide," General Research Corporation CR-1-722, April 1977.

5. C. Gannon, N. B. Brooks, JAVS Technical Report: RADC-TR-77-126, Volume II, "Reference Manual," General Research Corporation CR-1-722, April 1977.

6. Statement of Work for JAVS Implementation, PR B-6-3282, Rome Air Development Center, Griffiss Air Force Base, New York, October 8, 1975.

7. C. Gannon, JAVS Acceptance Tests for RADC, General Research Corporation IM-1998, October 1975.

8. N. B. Brooks, E. F. Miller, Jr., W. R. Wisehart, JAVS Final Report, General Research Corporation CR-4-465, December 1975.

9. N. B. Brooks, C. Gannon, JAVS Final Report, General Research Corporation CR-3-722, February 1977.

10. RXVP, FORTRAN Automated Verification System, Level 1, Reference Manual, General Research Corporation, May 1975.

11. R. J. Urban, L. W. Hambly, Extensible Automated Verification System (EAVS), Reference Manual, General Research Corporation CR-2-725, February 1977.

12. Advanced Quality Assurance Research Plan, General Research Corporation CR-4-720 (in preparation).

13. N. B. Brooks, Test Plan and Procedures for JOVIAL Automated Verification System (JAVS), General Research Corporation CR-2-465, July 3, 1975.

METRIC SYSTEM

BASE UNITS:

Quantity	Unit	SI Symbol	Formula
length	metre	m	...
mass	kilogram	kg	...
time	second	s	...
electric current	ampere	A	...
thermodynamic temperature	kelvin	K	...
amount of substance	mole	mol	...
luminous intensity	candela	cd	...

SUPPLEMENTARY UNITS:

plane angle	radian	rad	...
solid angle	steradian	*sr*	...

DERIVED UNITS:

Acceleration	metre per second squared	...	m/s
activity (of a radioactive source)	disintegration per second	...	(disintegration)/s
angular acceleration	radian per second squared	...	rad/s
angular velocity	radian per second	...	rad/s
area	square metre	...	m
density	kilogram per cubic metre	...	kg/m
electric capacitance	farad	F	A·s/V
electrical conductance	siemens	S	A/V
electric field strength	volt per metre	...	V/m
electric inductance	henry	H	V·s/A
electric potential difference	volt	V	W/A
electric resistance	ohm		V/A
electromotive force	volt	V	W/A
energy	joule	J	N·m
entropy	*joule per kelvin*	...	J/K
force	newton	N	kg·m/s
frequency	hertz	Hz	(cycle)/s
illuminance	lux	lx	lm/m
luminance	candela per square metre	...	cd/m
luminous flux	lumen	lm	cd·sr
magnetic field strength	ampere per metre	...	A/m
magnetic flux	weber	Wb	V·s
magnetic flux density	tesla	T	Wb/m
magnetomotive force	ampere	A	...
power	watt	W	J/s
pressure	pascal	Pa	N/m
quantity of electricity	coulomb	C	A·s
quantity of heat	joule	J	N·m
radiant intensity	watt per steradian	...	W/sr
specific heat	joule per kilogram-kelvin	...	J/kg·K
stress	pascal	Pa	N/m
thermal conductivity	watt per metre-kelvin	...	W/m·K
velocity	metre per second	...	m/s
viscosity, dynamic	pascal-second	...	Pa·s
viscosity, kinematic	square metre per second	...	m/s
voltage	volt	V	W/A
volume	cubic metre	...	m
wavenumber	reciprocal metre	...	(wave)/m
work	joule	J	N·m

SI PREFIXES:

Multiplication Factors	Prefix	SI Symbol
1 000 000 000 000 = 10^{12}	tera	T
1 000 000 000 = 10^{9}	giga	G
1 000 000 = 10^{6}	mega	M
1 000 = 10^{3}	kilo	k
100 = 10^{2}	hecto*	h
10 = 10^{1}	deka*	da
0.1 = 10^{-1}	deci*	d
0.01 = 10^{-2}	centi*	c
0.001 = 10^{-3}	milli	m
0.000 001 = 10^{-6}	*micro*	μ
0.000 000 001 = 10^{-9}	nano	n
0.000 000 000 001 = 10^{-12}	pico	p
0.000 000 000 000 001 = 10^{-15}	femto	f
0.000 000 000 000 000 001 = 10^{-18}	atto	a

* To be avoided where possible.

MISSION
of
Rome Air Development Center

RADC plans and conducts research, exploratory and advanced development programs in command, control, and communications (C^3) activities, and in the C^3 areas of information sciences and intelligence. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.